Kent Academic Repository

Full text document (pdf)

Citation for published version

Modi, Bala (2015) FPGA-based High Throughput Regular Expression Pattern Matching for Network Intrusion Detection Systems. Doctor of Philosophy (PhD) thesis, University of Kent,.

DOI

Link to record in KAR

https://kar.kent.ac.uk/56664/

Document Version

UNSPECIFIED

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version.
Users are advised to check **http://kar.kent.ac.uk** for the status of the paper. **Users should always cite the published version of record.**

Enquiries
For any further enquiries regarding the licence status of this document, please contact:
researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at **http://kar.kent.ac.uk/contact.html**

KAR Kent Academic Repository

University of Kent

FPGA-based High Throughput Regular Expression Pattern Matching for Network Intrusion Detection Systems

A thesis submitted to the University of Kent,
in the subject of Computer Science
for the degree of Doctor of Philosophy (PhD)

by
Bala Modi
February, 2015

To my wife Tina & children

Word Count: 61, 417

Abstract

Network speeds and bandwidths have improved over time. However, the frequency of network attacks and illegal accesses have also increased as the network speeds and bandwidths improved over time. Such attacks are capable of compromising the privacy and confidentiality of network resources belonging to even the most secure networks. Currently, general-purpose processor based software solutions used for detecting network attacks have become inadequate in coping with the current network speeds. Hardware-based platforms are designed to cope with the rising network speeds measured in several gigabits per seconds (Gbps). Such hardware-based platforms are capable of detecting several attacks at once, and a good candidate is the Field-programmable Gate Array (FPGA). The FPGA is a hardware platform that can be used to perform deep packet inspection of network packet contents at high speed. As such, this thesis focused on studying designs that were implemented with Field-programmable Gate Arrays (FPGAs). Furthermore, all the FPGA-based designs studied in this thesis have attempted to sustain a more steady growth in throughput and throughput efficiency. Throughput efficiency is defined as the concurrent throughput of a regular expression matching engine circuit divided by the average number of look up tables (LUTs) utilised by each state of the engine's automata. The implemented FPGA-based design was built upon the concept of equivalence classification. The concept helped to reduce the overall table size of the inputs needed to drive the various Nondeterministic Finite Automata (NFA) matching engines. Compared with other approaches, the design sustained a throughput of up to 11.48 Gbps, and recorded an overall reduction in the number of pattern matching engines required by up to 75%. Also, the overall memory required by the design was reduced by about 90% when synthesised on the target FPGA platform.

Acknowledgements

I am highly appreciative of my supervisor Gerald Tripp, for his professional assistance and guidance throughout the period of my research. I also wish to appreciate the effort of the entire School of Computing staff for their administrative assistance. I also thank the school for the technical and financial support I received during my study period. I thank all the members of my review panel. I also appreciate the support of my colleagues Keith Greenhow and Neil Brown for their support.

I wish to acknowledge the use of the Xilinx ISE Proprietary Project Navigator application version P.49d, volume 14.4 (nt64) Software. The software is bundled with the Xilinx Synthesis Tool (XST) used for synthesising and implementing the various circuit descriptions contained in the design that was constructed in this thesis. I acknowledge the use the Microsoft Excel 2010 spread sheet application for generating all the graphs and related figures and tables used for analysing the results of the experiments performed in this thesis. I also wish to acknowledge the use of the tool used for performing a one-way analysis of variance (One-way ANOVA). The ANOVA tool is bundled with the International Business Machines Statistical Package for the Social Sciences (SPSS) version 21 which was used for testing the statistical research hypothesis of the thesis.

Lastly, I wish to thank God almighty for his infinite love, protection, provision, wisdom and strength, as well as my wife Tina, and my children Yusuf, Stephen and Salim for their love and patience. I also extend the same appreciation to my aunty Esther, and my grandmother Jebu for their patience, financial, moral and spiritual support throughout my study period.

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ABS IDS	Anomaly-Based Intrusion Detection System.
AC-DFA	Aho-Corasick Deterministic Finite Automata.
ANOVA	Analysis of Variance.
ASCII	American Standard Code for Information Interchange.
BRAM	Block Random Access Memory.
CLB	Control Logic Block.
DFA	Deterministic Finite Automata.
ECD	Equivalence Class Descriptor.
ECD-NFA	Equivalence Class Descriptor Nondeterministic Finite Automata.
ECD_RTS-NFA	Equivalence Class Direct Synthesis Nondeterministic Finite Automata.
ECI	Equivalence Class Index.
FPGA	Field Programmable Gate Array.
FF	Flip-Flop.
FSM	Finite State Automata.
HDL	Hardware Description Language.
IBM	International Business Machines.
IDS	Intrusion Detection System.
LUT	Look up Tables.
MHz	Megahertz.
NIDS	Network Intrusion Detection System.
NFA	Nondeterministic Finite Automata.
OS	Operating System.
PCRE	Perl Compatible Regular Expression.
RE	Regular Expression.
RE-NFA	Regular Expression Nondeterministic Finite Automata.
RFC	Recursive Flow Classification.
REME	Regular Expression Matching Engine.
SBS IDS	Signature-Based Intrusion Detection System.
SPSS	Statistical Package for the Social Sciences.
UUT	Unit Under Test.
VHDL	VHSIC Hardware Description Language.
WWW	World Wide Web.
Xilinx ISE	Xilinx Integrated Synthesis Environment.

Glossary

Term	Definition
BGnRE	Block-Engine Group of n Regular Expressions, where n = 2, 3, 4 and 5.
BGnRE average ECDs	The average number of ECDs for each BGnRE REME.
BGnRE 4-byte CH	The sum of the characters matched per clock cycle by each 4-byte BGnRE REME, with n = 2, 3, 4, and 5.
Input	Number of characters consumed at once in bytes.
NOCH	Total number of characters matched.
NoL	Number of LUTs.
NoS	Number of NFA states.
Regexp	Regular Expression
T/Char	Total number of characters matched.
Tp	Throughput.
Tpe	Throughput Efficiency. It is the concurrent throughput of a regular expression matching engine circuit divided by the average number of look up tables (LUTs) utilised by each state of the engine's automata.

1. Introduction

This chapter briefly introduces the concept of computer networks and the related security issues involved. This is closely followed by the fundamentals, problem statement, thesis statement, contributions and finally the outline of the research work.

1.1 Introduction to Computer Network Security and Management

A computer network is defined to be a set of autonomous, independent computer systems which are interconnected so as to permit interactive resource sharing between any pair of systems" (Roberts and Wessler 1970). As such, there is a need to monitor how the various interconnected computer systems are secured on any given network.

Computer network security (Rufi 2006, p. 4) refers to the various processes involved in protecting computers on a network. The protection is against malicious (dangerous) attacks or intrusions by both external and internal users. Also, computer-mediated communication networks have succeeded in linking people together, spreading knowledge and connecting institutions together from all over the world. Networked societies regard such a network as a computer-supported social network (Wellman 2001, pp. 2031-2034). The printed media and publications such as newspapers, books, magazines, and posters to mention but a few have adapted to website technology. The technology uses a system of interlinked hypertext documents referred to as the World Wide Web (WWW or W3).

Furthermore, network management became necessary. The management process involves a broad range of functions that involve the use of tools to administrate, operate, and reliably maintain computer network systems. The deployment of security measures have to work in line with organisational measures and policies in order to be effective (Gollmann 2011). Also, network management requires the use of a variety of software and hardware devices, which aid in the network administration process. The management process is concerned with the reliability, efficiency, capacity and capabilities of data transfer channels. The process also covers a broad range of research components such as: security, performance and reliability. The security component deals with the protection and authorisation of eligible users on the network. Although, there may not be a clear distinction between the security-relevant components of a given system, the use of specific rules and regulations to check the excesses of legitimate users on the network should be encouraged.

The performance component deals with the ways to identify and fix bottlenecks in the network such as network connection and speed. The reliability component ensures an uninterrupted availability of

network service, and also ensures that the problems relating to hardware and software malfunctions (or crashes) are easily restored.

To capture the essence of computer security, the process of information asset compromise need to be understood as described by Gollmann (2011, p. 34) based on the following:

a. Confidentiality: This involves the prevention of unauthorised disclosure of information.

b. Integrity: This involves the prevention of unauthorised modification of information.

c. Availability: This involves the prevention unauthorised withholding of information or resources.

Most network security systems rely on layers of protection. The layers are made up of multiple components including network monitoring and security applications. The components are usually managed by a trained network or system administrator, who implements the security and policy requirements. The administrator is normally required to protect the confidentiality and integrity of secured information, while ensuring that only legitimate employees are granted adequate access to the network resources.

1.2 Fundamentals

This thesis is primarily concerned with developing a novel approach that is capable of effectively detecting attacks or intrusions flowing into a computer network. Such attacks usually appear in a given pattern within a data packet, and require to be stopped in real-time. The design described in this thesis is suitable for matching multiple characters and patterns at once. The process of performing the matching activity is known as pattern matching. Pattern matching as described by Komendantsky (2012, p. 150), involves the problem of deciding whether or not given a word defined over a finite alphabet of characters, and a given regular expression belongs to the language expressed by the regular expression. As such, given any T* representing the set of all strings defined over an alphabet T, then any 'regular expression defined over T always describes a regular language' (Rayward-Smith 1991). Refer to Section 2.2 for more background details on regular expressions.

Furthermore, any pattern that can be matched by a regular expression can also be matched by an automaton. Unlike pattern recognition which performs a likelihood of matching inputs, pattern matching is expected to generate an exact match. Parallel pattern matching involves the process of scanning multiple flows of characters with the aim of finding exact elements of a given pattern. Issues concerning network intrusions are security concerns that require urgent and proper attention. Intrusions are more commonly referred to as attacks or anomalies. Such attacks are actions or attempts aimed at compromising the confidentiality, integrity and availability of network resources.

Also, intrusions are more difficult to define in terms of their behaviour or actions, but could be easily expressed in terms of their effect on a given automaton. As such, intrusion detection can be regarded as a method by which intrusions from outside or misuse from within an organisation's network are gathered and analysed using an intrusion detection system (IDS). The detection system is aimed at stopping any possible security breaches from incoming or existing attacks. The primary objective of this thesis is to create a design that frames the intrusion detection problem as a pattern matching problem. The problem is solved using efficient algorithms to implement the matching processes (refer to Section 2.3 for more on IDSs). Furthermore, the entire research is aimed at developing a fast, more efficient and less complex design for performing pattern matching.

The choice of automata for the implementation of the pattern matching design depends on the appropriate choice of polynomial ($\mathcal{P}$) algorithm for deciding the basic problem. A polynomial time algorithm is any algorithm that runs in polynomial time O(n^k), where k some constant independent of the problem size n (Mount 2003, p. 57). Consequently, $\mathcal{NP}$ problems could be solved using a deterministic or nondeterministic Turing Machine (TM) (Hopcroft, Motwani and Ullman 2001, p. 414) respectively. Such $\mathcal{NP}$ problems are set of all decision problems (problems with Boolean outputs e.g. 'yes' or 'no') that can be verified in polynomial time (Mount 2003, p. 59). The design automata in this thesis runs in polynomial time O(n^km), with k being any positive integer, and n being the length of the regular expression. The value m is the number of regular expressions combined together to create the compound automata. Also, the memory requirement of the design runs in time O(nm). The choice of automata used in this thesis is based on the class of $\mathcal{NP}$ problems under consideration. The class of such problems can best be solved using nondeterministic TMs as opposed to the deterministic TMs (Hopcroft, Motwani and Ullman 2001, p. 414).

A number of optimisation strategies are employed in this thesis in order to generate a more efficient automata-based design. Such strategies are usually implemented as separate approaches, but this thesis combined multiple strategies into a single design. The design is then used for creating compact and efficient pattern matching systems that operate in a parallel configuration. Such optimisation strategies are discussed in Section 3.2. The idea is to generate a memory efficient and fast nondeterministic finite automata (NFA) based matching engines. The automata are finally implemented in a target Field Programmable Gate Arrays (FPGA) platform to perform hardware-based regular expression matching. Section 3.2.2f dwells more specifically on the concept of equivalence classification used to develop the approach in this thesis.

An analysis of the various related approaches is performed in Chapter 4 to measure how each approach compares to the other. After introducing the thesis approach in Chapter 3, the analysed results in Chapter 4 was collated and combined with the results of experiments obtained in the thesis in Chapter 6. Afterwards, a comprehensive analysis was performed again and further results were obtained.

1.3 Motivation

As network bandwidth has continued to steadily and rapidly increase, so have the frequency of network attacks. Network bandwidth is considered to be a measurement of the bit rate that a network interface supports, and is usually expressed in Bits/sec, Kilobits/sec (Kbps), Megabits/sec (Mbps), and Gigabits/sec (Gbps) etc.

Data packets need to be processed and as such packet processing has become necessary. Furthermore, most patterns contained in data packets now appear in form of complex regular expression patterns which are increasingly difficult to define and detect. Also, string matching is no longer efficient in defining and detecting such complex patterns. Such patterns are mostly contained in packets of data that flow through a network system. Furthermore, the required processing flexibility and fast processing power could best be attained by utilising reconfigurable hardware devices such as FPGAs. Current FPGAs exploit specialised circuitry and parallelism, and have the ability to process multiple bytes or characters per clock cycle.

Currently, many approaches have been developed and implemented to deal with the problem of network intrusions using Network Intrusion Detection Systems (NIDS). The use of regular expression pattern matching as a viable alternative and solution to string pattern matching has become necessary. This is because regular expression matching systems have the ability to effectively scan a packet payload in order to detect suspicious contents. A packet payload is the data in the body of the packet block during a data transmission process. However, regular expressions are often a computational burden in applications such as NIDS, which rely heavily on the regular expressions. The computational burden is high, because matching against regular expression consumes a lot of computation time, which is in O(n^km), with k being any positive integer, and n being the length of the regular expression. The value m is the number of regular expressions combined together to create the compound automata.

The key challenge facing the development of an efficient regular expression pattern matching design lies in the ability to take full advantage of the processing speed and fine-grained parallelism provided by current reconfigurable hardware platforms such as FPGAs. The speed and parallelism is the fundamental advantage of FPGAs over microprocessor-based regular expression matching designs. Almost all the previous approaches have been trying to deal with issues such as: the increased clock operating frequency (or simply the design speed of matching measured in MHz) at a much reduced logic circuit cost. The logic circuit cost relates to the resources utilised within a confined FPGA logic space. Other issues include reduction in the latency and the number of logic elements utilised by the states of a given automata. Latency can be described as the delay experienced during a certain processing stage, such as the latency experienced in a parallel input/output (I/O) system during disk access operations. The issue of power consumption could also be efficiently dealt with by applying various optimisation strategies used for the effective hardware implementation.

However, the primary concern in this thesis is to provide a good balance between obtaining a higher throughput, which is the continuous rate at which an input data stream is processed (Brodie, Taylor and Cytron 2006), and a higher throughput efficiency (Yang, Jiang and Prasanna 2008). The throughput efficiency determines the efficient use of logic resources by every state of a given FPGA-based automata such as the use of look up tables (LUTs) (refer to Section 2.6 for more details on LUTs). Furthermore, computing the throughput efficiency is important. This is because the throughput of a regular expression matching circuit alone is not enough to justify the length and complexity of the regular expressions implemented (Yang and Prasanna 2012). Also, the LUT efficiency alone does not account for the clock frequency of the matching circuit. This explains why computing the throughput efficiency is important in determining the efficiency of the designs implemented in this thesis. Refer to Section 6.2.3 for more on the computation of the throughput efficiency.

1.4 Thesis Statement

This thesis implemented a design that performs regular expression pattern matching using FPGAs. The approach addresses the problem of creating more compact and efficient VHSIC Hardware Description Language (VHDL) designs files. The files contain the memory and table-synthesis modules, which are attached to the Equivalence Class Descriptor (ECD) NFA-based automata simply referred to as ECD-NFA. The ECD-NFA design was further improved by developing the optimised version of it called an Equivalence Class Direct Table-Synthesis NFA referred to as ECD_RTS-NFA. The ECD_RTS-NFA

completely discards the need for decoder circuits, together with the associated shift registers, multiplexers and other related logic circuit components. Chapter 5 discusses the full design and implementation of the approach. Furthermore, the approach implemented in this thesis incorporated the basic functionalities of a typical Snort NIDS. Snort was originally released in 1998 by Sourcefire founder, Martin Roesch (Roesch 1999, p. 229). Typically, a Snort NIDS is based on rules and those rules are also based on intruder signatures. Section 2.2.1 discusses more about the structure of a Snort rule.

Perhaps one of the most important aspects of this thesis which required attention was how to potentially reduce the large state transition table size of the ECDs. Problem of large input table size is an issue with most Deterministic Finite Automata (DFA) and other poorly designed NFA approaches. This thesis succeeded in reducing the hardware input table size by synthesising the table into a piece of logic using the Xilinx Synthesis Technology (XST) tool (Xilinx ISE 2012). The XST VHDL synthesis tool is bundled in the Xilinx Integrated Synthesis Environment (ISE) FPGA Project Navigator, version 14.4 design suites. The synthesised table is then used to perform the necessary table lookup operations, which could be inefficient if not carefully designed. To improve the design, the table lookup operation function avoided the use of complex nested loop operations (refer to Section 5.3.3 for more details). This is simply because complex nested lookup operations are not supported by FPGAs.

The design also addresses the current trend in pattern matching called parallel multi-character (Singapura et al. 2015) and multi-pattern matching. These types of pattern matching systems involve the creation of many blocks of NFA-based automata that perform regular expression matching in parallel. These blocks are called regular expression matching engines (or simply referred to as REMEs throughout this thesis). REMEs are usually built for a single pattern matching engine (Lin et al. 2006; Hieu et al. 2011) in most related approaches. The design implemented in this thesis combined several patterns of varying degrees of complexity together within the same REME block. More importantly, the concept placed four sub-REMEs within every REME block, which were then arranged them in a parallel configuration. The design approach is fully explained Chapter 5.

Lastly, the approach described in this thesis ensures that the throughput of the design only decreases steadily with every increase in the REME design complexity. The efficiency of the throughput decreases steadily too, especially in terms of the how much memory it consumes. The time it takes to synthesise, place and route the entire design on hardware is also reduced to the barest minimal. The performance of the overall throughput and throughput efficiency of the various REME designs in this thesis forms the basis for the statistical research hypothesis stated in Section 1.6.

1.5 Research Objective

The summary of the research objective is to determine whether or not the objective of creating scalable REME designs is achievable. The idea is to ensure that the REME designs only steadily decline in throughput and throughput efficiency. The rate at which this happens is to be established based on the result of experiments. Each of the studied approaches in this thesis that tried to scale up their REME designs also reported declines in their design throughput and throughput efficiency. Also, only a few of the directly related LUT-based approaches reported the number of LUTs utilised by their designs. A LUT is a collection of logic gates (National Instruments 2011) hard-wired on the FPGA. LUTs are used to implement function generators in CLBs (Xilinx 2012a). However, the number of LUTs utilised by each state of automata in a given sub-REME design automata is central to the computation of the throughput

efficiency of the design. Four sub-REME matching blocks make up a REME in this thesis. Lastly, the focus in this thesis is mainly on the LUT-based REME design approaches as discussed in Chapter 6.

1.6 Statistical Research Hypothesis

The experimental hypothesis of this thesis seeks to address the objective mentioned in Section 1.5. The hypothesis was then tested for significance in Section 6.3 of Chapter 6 and is stated thus:

a. **Null Hypothesis (H_O)**: Increasing the number of patterns matched by each sub-REME has no effect on the throughput and the number of LUTs utilised by each sub-REME in the overall REME.
b. **Alternative Hypothesis (H_A)**: Increasing the number of patterns matched by each sub-REME has an effect on the throughput and the number of LUTs utilised by each sub-REME in the overall REME.

1.7 Contributions

The details of the contributions made by this thesis are as discussed in Section 7. However, this section highlights the summaries of the contributions made as follows:

i. The thesis introduced a novel ECD-NFA two-phased approach (refer to Section 5.4 for more details) with its optimised version called the ECD_RTS-NFA. The approach is used to generate efficient NFAs for the various REMEs.
ii. The process involved in performing (i) uniquely applies to NFAs only. This is because with NFAs, there is not necessarily a single current state that is reached anymore. Furthermore, a single ECD input can activate several transitions and states which is not applicable to DFAs.
iii. All the inputs that do not trigger transitions were properly classified. This was achieved by grouping the inputs as a single ECD descriptor to be looked up. Also, all self and empty string transitions were also eliminated in the process, thereby further reducing memory requirement.
iv. A minimised and compressed n-dimensional table of compressed ECDs (Gupta and McKeown 1999, p. 150) was created. The table is suitable for a 4-byte input match.
v. A simple algorithm was implemented to synthesise the table of ECDs generated in (iv) into a simple piece of logic for look up operations. This helped to cut down a lot of logic resources required such as: shift registers and decoders.
vi. A very simple and less complex toolchain for implementing simple, fast and area efficient NFA-based REMEs was implemented. The process was divided into the first phase, which is the software implementation phase, and the second phase which is the hardware implementation.
vii. The design comprehensively combines multiple optimisation strategies discussed in Chapter 3. The design uses equivalence classification concept to create compact, memory efficient and fast NFAs for the hardware implementation in the second phase. The design also matches multiple characters by consuming four characters at once. The approach is designed to increase the speed of matching characters.
viii. An algorithm was implemented to build nested sub-REME (Yang and Prasanna 2008; Yang and Prasanna 2012) blocks into each REME mentioned in (vi). The blocks are then arranged in parallel to execute multiple patterns in a single clock cycle.

ix. Using optimised the ECD_RTS-NFA, a unique form of block RAM (BRAM) compression was utilised for the ECDs generated in the software design phase. The BRAMs supply 7-bit ECDs to the various matching units. This is in contrast to the BRAM centralised character matching approach (Xilinx 2011; Yang, Jiang and Prasanna 2008; Yang and Prasanna 2009; Ganegedara, Yang and Prasanna 2010; Hieu et al. 2011; Long et al. 2011) implemented for character matching.

The main purpose of this thesis was to create an all-round design that combined numerous design strategies into a single approach. The design utilised the concept of equivalence classification for an NFA-based design, previously known to work with DFAs only as described by Brodie, Taylor and Cytron (2006, p. 194) and Tripp (2006). The concept was used to implement a variation of the DFA-based approach that now works with NFAs only. The detailed design process is found in Chapter 5, while the detailed analysis of the design performance is found in Chapter 6.

1.8 Outline

Chapter 1 briefly discusses the concept of computer network security and management. It also highlights some of the reasons that prompted research in the area of regular expression pattern matching for implementation in current NIDSs. The reasons leading to the novel approach described in the chapter is briefly introduced, while the various contributions made in thesis is highlighted. The chapter concludes with the statement of the research hypothesis.

Chapter 2 gives brief background knowledge behind regular expression, and examines the issues relating to security breaches in current NIDS. The security threats and weaknesses used to exploit most computer networks are discussed. The chapter further elaborates on the basic concepts concerning Finite Automata (FAs). Some existing hardware platforms are briefly introduced, with a major focus on FPGAs. Hardware description languages (HDLs) were also briefly introduced, with a primary focus on VHDL, which is the HDL of choice in this thesis.

Chapter 3 discusses the related approaches used in designing efficient regular expression pattern matching engines. The approaches are explained and closely examined in order to understand the strategies used in their implementation. The idea is to explore and relate them to the initial novel ECD-NFA approach, which is later optimised as ECD_RTS-NFA as described in Section 1.4.

Chapter 4 relates to Chapter 3, but mainly analyses the various results obtained from all the related approaches discussed in Chapter 3. The analysis centres on the way each approach compares to the rest. The comparison is mainly concerned with the number of bytes consumed per clock cycle, as well as the density of the design. The density reflects the amount of logic resources utilised per state of a given automata. The throughput generated by the separate approaches and the number of patterns matched also forms another basis for comparison.

Chapter 5 describes in detail the FPGA-based approach described in this thesis. The approach in the chapter describes the novel two-phased design and toolchain approach. In the toolchain, the first phase describes the software based modular block module that is used to automatically generate the ECD_RTS-NFA VHDL codes necessary for interfacing with the second phase. The second phase of the approach describes the hardware-based modular block module, which is made ready for hardware synthesis, mapping, placement, and routing.

Chapter 6 analyses and explains the results obtained from the design described in Chapter 5. An evaluation of the various results obtained for the 4-byte input matching REME engines is presented. The chapter analyses the results obtained in this thesis against those obtained in Chapter 4. The analysis aims to establish the performance of the thesis design, while identifying its limitations too.

Chapter 7 outlines the summary of the various contributions made by this thesis approach, and the conclusions drawn from it. The chapter also outlines the possible future work required to build upon the current design. A recommendation for improving some of the related approaches is highlighted. The chapter ends with a brief evaluation and concluding thoughts concerning the future of high-speed NIDs.

A brief background of the study in this thesis is discussed in Chapter 2. Emphasis is on the basic concepts relating to regular expressions and how each relates to the design automata which are NFAs. The hardware platform of choice in this thesis, as well as the hardware description language used for implementing the various REMEs is further discussed in Chapter 2.

2. **Background**

Chapter 2 discusses the background knowledge of strings, regular expression and pattern matching. It also discusses the issues affecting network security and its accompanied threats and weaknesses. This chapter also discusses the hardware technologies used for pattern matching and the associated finite state machines. The hardware description language used to implement the various REMEs is also discussed in this chapter.

2.1 Introduction

The regular expression pattern matching engines described in this thesis are designed and implemented for FPGA platforms. This chapter starts by explaining the basic concepts relating to regular expressions. Section 2.2.1 illustrates the format in which regular expressions exist within a given Snort rule, which are usually in form of Perl compatible regular expressions (PCREs). Afterwards, using some basic operations, a description of simple NFAs constructed from regular expressions is discussed. The basic operations allow for the construction of more complex NFAs. The regular expressions described in this chapter appear in most of the rules found in several NIDS. The rules constitute the patterns of attacks present in most computer networks. The chapter further discusses what constitutes an NIDS and the security issues involved. The Hardware Description Language (HDL), basic FPGA logic circuits and the synthesis process involved is also discussed.

2.2 Regular Expression

To fully understand the concept behind regular expression pattern matching and the various approaches that implement them, there is a need to first understand what constitutes regular expressions. This Section summarises the concepts of alphabets, strings, and regular languages which constitutes regular expressions.

An alphabet is a finite non-empty set of symbols. Conventionally, the symbol Σ is often used to represent an alphabet (Hopcroft, Motwani and Ullman 2001, pp. 28-29). Examples of an alphabet include:

i. $\Sigma = \{0,1\}$, which is a binary alphabet.
ii. $\Sigma = \{a,b,....,z\}$, which is a set of all lower-case letters.
iii. A set of all American Standard Code for Information Interchange (ASCII) characters, which are printable.

A string (or word) is a finite sequence of symbols chosen from some alphabet Σ. In a binary alphabet such as Σ = {0,1}, a string of binary numbers could be formed, such as 1110001. An empty string is denoted by ϵ [which] may be chosen from any alphabet whatsoever. A standard notation for determining the length of a given string x is given by $|x|$. For instance: $|abb|$ = 3, $|x|$ = 1, and $|\epsilon|$ = 0. Furthermore, the notation defined as Σ^t is normally defined to be the set of all strings of length t, with each having its symbol in the alphabet. Thus, $\Sigma^0 = \{\epsilon\}$, regardless of what Σ represents. If Σ = {a,b} then Σ^1 = {a,b} and Σ^2 = {aa, ab, ba, bb}.While Σ and Σ^1 may contain the same elements, Σ is an alphabet while Σ^1 is a set of strings with length of 1 each. The set of all strings defined over an alphabet is denoted by Σ^*. Thus, given that Σ = {x,y}, Σ^* = { ϵ , x, y, xy, yx, xx, yy, ….} was obtained. Σ^* could still be expressed further as: $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup$ .....

A language is a finite set of all strings chosen from some Σ^*. For the sake of clarity, if Σ is an alphabet, and $L \subseteq \Sigma^*$, then L is considered to be a language over Σ. It is also important to note that:

i. Σ^* is a language for any alphabet Σ.
ii. Ø is the empty language over any alphabet Σ.
iii. $\{\epsilon\}$ is the language composed only of empty strings.

However, while Ø have no strings at all, $\{\epsilon\}$ stands for a single string of length 0. Thus, $Ø \neq \{\epsilon\}$, $Ø^* = \{\epsilon\}$ and $Ø^0 = \{\epsilon\}$, $Ø^i \forall, i \geq 1$ is empty because no strings can be selected form an empty set (Hopcroft, Motwani and Ullman 2001, p. 85). There are three basic operations that could be performed on a language, namely: union, concatenation and Kleene closure. The closure of a given language L is denoted L^*, represents a set of strings that can be formed by taking any possible number of strings from L. The term "Kleene closure" is attributed to S.C Kleene who is the originator of the regular expression notation and the closure operator (*). Given any two languages L and M, such that L = {aab, ba, bbb}, and M = {ϵ, aab}, then Hopcroft, Motwani and Ullman (2001, pp. 85-86) describe the following operations as:

i. L∪M = {ϵ, ba, aab, bbb}, which represents the union of L and M.
ii. LM = {aab, ba, bbb, aabaab, baaab, bbbaab}, which represents the concatenation of L and M.
iii. M^* = {ϵ, aab, aabaab, aabaabaab…}, which represents the Kleene closure of M.

A finite language is considered to be a regular language, and the following definitions hold for a regular language:

i. If a language is empty, it is considered to be a regular language. Thus, Ø is a regular language.
ii. If a $\in \sum$, then the language $\{a\}$ is a regular language.
iii. If L and M are regular languages, then L∪M (union), LM (concatenation), and M^* (Kleene closure) are all regular languages too.

A regular expression [can be used to] define exactly the same regular languages that various forms of the automata describe (Hopcroft, Motwani and Ullman 2001, p. 83). Section 2.2 discusses more on such automata. Furthermore, given that any T* represents the set of all strings defined over the alphabet T, 'then any regular expression defined over T always describes a regular language' (Rayward-Smith 1991). Also, a regular expression (which is more commonly referred to as a 'regexp') is a regular language constructed with character classes over a fixed alphabet (Yang, Jiang and Prasanna 2008). Regexps are commonly used to serve as the input language for a lot of systems concerned with string processing. In addition to the three basic operations carried out on a regular language, namely: union, concatenation, and Kleene closure; there are other common operations. Algebra allows for the construction of some form of

simple expressions, by repeatedly applying a set of arithmetic operators such as: + (plus) and * (multiplication) to previously constructed expressions. Hopcroft, Motwani and Ullman (2001, pp. 86-87) discussed the inductive steps involved in regexps operations as follows:

1. If a and b are regexps, then a + b is also a regexp that denotes the union of L(a) and L(b), expressed as L($a + b$) = L(a) ∪ L(b).
2. If a and b are regexps, then ab is also a regexp that denotes the concatenation of L(a) and L(b), expressed as L(ab) = L(a) L(b).
3. If a is a regexps, then a^* is also a regexp that denotes the Kleene closure of L(a) expressed as L(a^*) = (L(a))*.
4. If a is a regexps, then (a) which is a parenthesised a, is also a regexp that denotes the same language as a, expressed as (L(a)) = L(a).

In terms of operator precedence, the Kleene closure operator has the highest precedence, followed by the concatenation operator and finally the union operator. A good example of a regular expression is given thus: "/(m|n)*(pq)/". The character "|" is called a pipe. When it is grouped in a regexp, it allows alternation of either parts of the regexp. For instance in (m|n), either "m" or "n" can be matched at a time. The Kleene closure operator in (m|n)* means that zero or more strings of either m or n is matched. Lastly from the regexp example "/(m|n)*(pq)/", zero or more strings of either m or n is matched, before a single p followed by a single q is matched.

It is importance to understand the general concepts behind regexps, before looking at how it applies to automata such as DFAs and NFAs. However, since the rules used for inspecting data packets flowing through a network contain harmful contents usually in form of regexps, there is a need to examine such rules. Section 2.2.1 discusses more on the structure of a typical Snort rule.

2.2.1 Snort Rule Description

Data is constantly flowing through a computer network in form of packets at any given time, creating traffic. Network traffic refers to the amount of data passing through such a network in a particular period of time. Contained in each of the data packets trafficking through the network are packet payloads. Each packet payload contains contents that may be harmful to a network system. The design in this thesis examines the packet payloads for suspicious contents that appear in the form of regexp patterns. The patterns are extracted from the Snort rules databases and closely examined.

Most intruder activities on a given network have similar signatures to those of viruses (refer to Section 2.4.1 for more on viruses). Also, the Snort rules considered in this thesis were extracted from the Sourcefire community rules v.2.0 (Rehman 2003; Snort 2013). The rules are created based on the captured information regarding the signatures of the intrusions. Furthermore, the existing databases of known vulnerabilities are constantly exploited by intruders. However, the attack signatures that are already known are used to determine when an intruder is attempting to exploit the databases of the known signatures.

Typically, Snort rules are composed of two logical parts namely: the rule headers and the rule options. The rule header and its options make up the structure of any given Snort rule as seen in Table 2.1.The rule header determines the action to be taken by a rule, as well as the criteria used to match a rule against a data packet. Alert messages and other information pertaining to that part of a packet used to

generate alert messages are contained in the options part of the Snort rule. The focus of Section 2.2.1 is to describe the PCRE patterns contained in the content part of the rule options. This is because this thesis is more concerned with automating the regular expressions found within each packet payload. The automaton is then used to perform a packet inspection of harmful data packets flowing through a network. Table 2.1 shows the general structure of a Snort rule, which is further described with an illustration of the Snort rule presented in Table 2.3. Afterwards, the keyword "pcre", which is the focus of this section, is explained separately in more detail.

Table 2.1: The general structure of a Snort rule (Rehman 2003, p. 79).

Rule Header	**Rule Options**

For illustrative purpose, consider the rule header and its options in Table 2.2. The rule was selected from the community service rule database provided by Sourcefire Inc. (Snort 2013).

Table 2.2: A typical Snort community service rule (Snort 2013).

	A typical Snort Rule
Sample Rule	alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"PROTOCOL-FTP SITE EXEC attempt";flow:to_server, established; content: "SITE"; nocase; content: "EXEC"; distance: 0; nocase; **pcre**:"/^SITE\s+EXEC/smi"; metadata: ruleset community, service ftp; reference: bugtraq,2241; reference: cve, 1999-0080; reference: cve,1999-0955; classtype: bad-unknown; sid: 361; rev: 22;)

Furthermore, the summary of the keywords found in the rule header of the Snort rule in Table 2.2 is presented in Table 2.3.

Table 2.3: Table of Snort rule header (Snort 2013; Sourcefire 2013).

Rule Header Keywords	**Description**
Alert	An alert is the first part of the Snort rule and it is generated and sent to a file or console whenever a rule condition is met. The alert rule action is one of the predefined actions within the Snort rules, and can be user-defined
Protocol: tcp	The Transfer Control Protocol (TCP) is one of the protocols found in the rule header. The TCP keyword checks to ensure that the receiving host is prepared to receive TCP-type data packets.
Source address: $EXTERNAL_NET	The source address variable is the $EXTERNAL_NET, and it defines the external and untrusted network addresses that connects to the home network.
Destination address: $HOME_NET	$HOME_NET is the destination address variable. It defines the home network addresses to be protected against.
Any	The keyword "any" following the $EXTERNAL_NET or $HOME_NET variable applies the rule on all packets irrespective of the port number.
21	The port number 21 after the $HOME_NET variable represents the port for all FTP-type packets.

The other keywords found in rule options of the Snort rule in Table 2.2 are described in Table 2.4.

Table 2.4: Table of Snort rule options (Snort 2013; Sourcefire 2013).

Rule Options Keywords	**Description**
msg	The "msg" keyword is used to add text string to logs or alerts, and the messages that follow the keyword are placed within double quotation marks. In the message "PROTOCOL-FTP SITE EXEC attempt" for instance, the File Transfer Protocol (FTP) server command executes commands on an FTP server. "EXEC" is the argument to the FTP command name "SITE".
Direction operator: -> or <-	The direction operator "->" shows that the address variable $EXTERNAL_NET and the port number "any" on the left hand side are the source, while the address variable $HOME_NET and the FTP port number 21 on the right hand side are the destination.
flow	Options can be used with the "flow" keyword to determine the direction of packet flow. The option "to_server, established;" establishes a TCP session. The keyword signifies a response when applying a rule during a TCP session.
content	This is a significant keyword, because of its ability to locate a data pattern within a data packet. The keyword is used to identify intrusive signature similar to malware such as viruses. For instance, the Snort rule detects the pattern "SITE" in the TCP packets leaving the address variable $EXTERNAL_NET and entering the home network address variable $HOME_NET to port 21 the port for FTP-type packets.
nocase	Is a keyword that is used in conjunction with the keyword "content". For instance. nocase; content: "EXEC", allows for a case insensitive search of the content pattern "EXEC" . The keyword "nocase" has no argument of its own.
distance	The keyword "distance" means that the next match starts relative to the end of a previously searched pattern when searching for a specific pattern. For instance in the command "SITE EXEC", the number of spaces to be ignored between "SITE" and "EXEC" is determined by the distance keyword of value 0. The distance of value 0 means start immediately after the previous pattern is matched. The allowable values for the keyword ranges from -65535 to 65535.
metadata	The "metadata" tag allows extra information concerning a rule to be embedded by rule writer based on a key-value format. For instance, "metadata: ruleset community, service ftp;" is a free-form metadata having multiple keys: ruleset and service, as well having multiple values: community and FTP.
reference	The keyword "reference" allows references to external attack identification systems to be included within the given Snort rule, including unique Universal Resource Locators (URLs). For instance, "reference: cve,1999-0955;" has a reference id system of type "cve" which is a URL prefix to the URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=, and reference id is given as: 1999-0955.
classtype	Snort rules use the "classtype" keyword for classifying attacks as part of a general attack class. This helps to organise the event data which Snort provides. For instance, "classtype: bad-unknown;" is a potentially bad traffic with medium priority.

Table 2.4: (*cont'*d).

Rule Options Keywords	**Description**
sid and rev	The keyword "sid" is a rule option that is usually used with the "rev" keyword. The "sid" keyword provides information that is used to uniquely identify Snort rules, while the "rev" keyword identifies the revisions of unique Snort rules. For instance, "sid: 361; rev: 22;" indicates that 361 is the unique id of the Snort rule and the revision is number 22.

The only rule option that was left out of the sample Snort rule shown in Table 2.2 is the "pcre" keyword. All the other Snort rule options have been discussed in Table 2.3 and continued in Table 2.4. The "pcre" allows PCRE rules to be written more efficiently. The PCRE has a library of functions suitable for implementing regexps pattern matching. As such, considering the sample Snort rule option pcre: "/^SITE\s+EXEC/smi", it can be clearly seen that its argument is a regexp. The description of the regexp simply means: match the command pattern "SITE" at the very beginning of the match string, followed by a single or multiple spaces before the "EXEC" command is matched. The pattern modifiers "smi" imply the following:

a. 'i': The modifier will make all the matched characters in the pattern case-insensitive, whenever it is set.
b. 's': If set, will make a dot metacharacter to match all the characters present in the pattern including newlines.
c. 'm': When set, ^ (caret) and $ (dollar) matches at the beginning and ending of the string. By default, the string is treated as a big line of characters.

Furthermore, pcre evaluation can be a computational burden especially when every data packet flowing through the network has to be evaluated (Sourcefire 2013). Handing over such a computational burden to a separate pattern matching system such as the one implemented in this thesis is helpful. However, there is still a need to understand how the regexps extracted from the various Snort rules discussed in Section 2.2.1 are converted to automata. Section 2.3 discusses more on what constitutes the Finite Automata (FA) as well as the basis and inductive steps involved in a regexp-to-NFA conversion, particularly an ϵ-NFA.

2.3 Finite Automata

A Finite Automaton (FA) is considered to be an abstract machine, which can be in one or several states at any given time. Hopcroft, Motwani and Ullman (2001) describe a finite automaton as one that comprises of a set of states, having a control which moves from one state to another in response to external inputs. Such a control could be "deterministic", meaning that a machine can only be in one state at any given time. If the machine has to be in several states at a given time, then the machine's control is "nondeterministic". There two basic types of FAs are DFAs and NFAs.

A DFA is a Finite State Machine (FSM) that accepts or rejects a finite string of characters or symbols. A DFA can be implemented in both hardware and software. Hopcroft, Motwani and Ullman (2001) describe the formulation for a given DFA say A as follows:

Given that A = (X, Σ, δ, q_0, F) a 5-tuple, then:

1. X is a finite set of states.
2. $\sum$ is a finite set of input symbols.
3. A transition function given as: $\delta : X \times \Sigma \rightarrow X$.
4. Start state $q_0 \in X$.
5. A set $F \subseteq X$ of accepting states.

Given that a DFA "A = (Q, Σ, δ, q_0, F), the language is denoted by L(A) and is defined by L(A) = {w | δ (q_o, w) is in F}" (Hopcroft, Motwani and Ullman 2001, p.52). The language of the DFA A consists of the set of strings w which take the start state q_o to one of the accepting states in F. The symbol δ is the transition function. Furthermore, "if L is L(A) for some DFA A, then…[we could say that] L is a regular language" (Hopcroft, Motwani and Ullman 2001, p. 52).

The DFA can be described using a directed graph. Each vertex of the graph represents a state, and edges represent possible transitions. The DFA is an FA which accepts or rejects finite strings of symbols. For each given input, there is one and only one state to which the DFA can make a transition from its current.

The NFA like the DFA is also described using a directed graph, and is considered to be a special kind of FA that accepts or rejects finite strings of symbols too. From each state and given input, the NFA can jump into several possible next states, when an input string of finite length is read by it. However, despite the added flexibility of an NFA over a DFA, an NFA cannot recognise a language that is not already recognised by a DFA. The proof that a DFA can do what an NFA does involves a special process of construction called the subset construction. The subset construction process constructs all subsets of the states of the NFA. Generally, most of the proofs concerning automata require constructing one automaton from another.

The smallest DFA in worst case can have up to 2^n states, while the smallest NFA has just n states for the same language recognised by the DFA (Hopcroft, Motwani and Ullman (2001, p. 61). The initial state of the NFA is designated by a start state that has an inward arrow attached to it. Without a source vertex, the machine begins to process strings from the state called the initial state, by reading the first symbol of the input string. Based on its value, it makes the appropriate transitions to the next states. States shown with a double boundary are called accepting states, and there can be many of these in an NFA. The remaining states that are neither the start nor accepting states are the intermediary states.

The formulation for a given NFA A is the same as that of a DFA, except for the transition function definition. The transition function for a given NFA A = (X, Σ, δ, q_0, F) is a 5-tuple such that its transition function is: $\delta : X \times \Sigma \cup \{\epsilon\} \rightarrow$ pow(X). Note that pow (X) denotes the power set of X, which contains the set of all subsets of X, including the empty set Ø, and X itself. Also, if an NFA A = (Q, Σ, δ, q_0, F), then L(A) = {w|δ (q_o, w) ∩ F ≠ Ø. It then follows that the L(A) is the set of strings w in Σ^* such that δ(q_o, w) contains at least one accepting (Hopcroft, Motwani and Ullman 2001, p. 59).

Moreover, to successfully convert regexps to automata, NFAs with ϵ-transitions which Hopcroft, Motwani and Ullman (2001, p. 72) calls an ϵ-NFA, were implemented in this thesis. The idea behind the ϵ-transition is that it allows an automaton to make transitions on the empty string ϵ. However, the

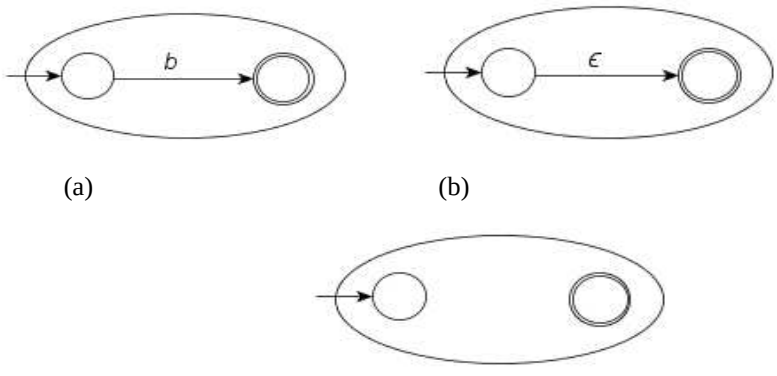
definition of acceptance of strings and languages by an ϵ-NFA is based on the formal definition of an extended transition function for the automata. By definition, a state q is ϵ-closed by following all the transitions leaving state q with the label ϵ. On reaching every state by following ϵ, the ϵ-transitions leaving those states are also followed until all the states reachable from state q along the paths labelled ϵ are found. However, the full discussion on the ϵ-closure of a state on ϵ-NFAs is beyond the scope of this thesis.

The ϵ-NFAs have a close relationship with regexps such that, when trying to prove the equivalence between the classes of languages accepted by both finite automata and regexps, the ϵ-NFA proves to be useful. The formal notation of an ϵ-NFA retains all the components together with the interpretations belonging to a traditional NFA. The only difference is that that the transition function δ now takes as arguments:

a. A state is in Q, and
b. A member $\Sigma \cup \{\epsilon\}$, which is either an input symbol or the symbol ϵ, and it is required that ϵ cannot be a member of the alphabet $\sum$ to avoid any confusion.

It then follows that by using ϵ-transitions it is possible to build an ϵ-NFA from a collection of regexps. This is achieved by introducing new ϵ-transitions from the newly created start state, to all the start states of all the ϵ-NFAs constructed from the various regexps. New ϵ-transitions are also introduced from their accepting states to an indistinguishably combined accepting state, which is possible as each of their various final states represents one output signal when implemented on a hardware device.

Figure 2.1 illustrates the basis for the construction of an automaton from a given regular expression. From Figure 2.1, the first figure labelled (a) represents the regexp b, and the language of its automaton is {b}. The figure labelled (b) shows that the language for the automaton is $\{\epsilon\}$, because there is only one path from the start state to the accepting and it is labelled ϵ (Hopcroft, Motwani and Ullman (2001, pp. 102-103). The last figure labelled (c) shows that Ø is the empty language for the automaton, which has no paths from the start to the accepting state. Figure 2.2 illustrates the inductive step for the regexp-to-ϵ-NFA construction, given the regexps r and s.



(a) (b)

(c)

Figure 2.1: Figure (a), (b) and (c), show the basis for the construction of an automaton from a regexp (Hopcroft, Motwani and Ullman 2001).

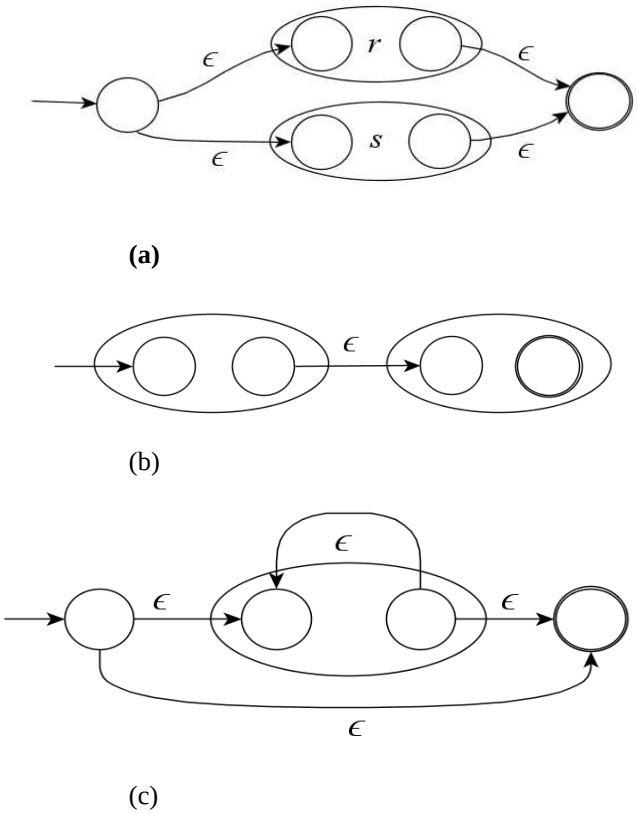


Figure 2.2: The inductive steps in the construction of regexp-to- ϵ-NFA construction (Hopcroft, Motwani and Ullman 2001).

From Figure 2.2, the first figure labelled (a) indicates that for the regexp r + s, the corresponding automaton starts from the new start state and then goes to either the start state of the automaton for r or that of s. Afterwards the accepting state of the automaton is reached by following the path labelled by the string L(r) or L(s) respectively (Hopcroft, Motwani and Ullman 2001, p. 103). Recursively applying the constructions in Figure 2.1 and 2.2 leads to the construction of the NFA for the regexp ((a|b)*)(cd) shown in Figure 2.3. Figure 2.3 shows how the NFA matches the set of strings: "aaaacd", "bbbcd", and "cd":

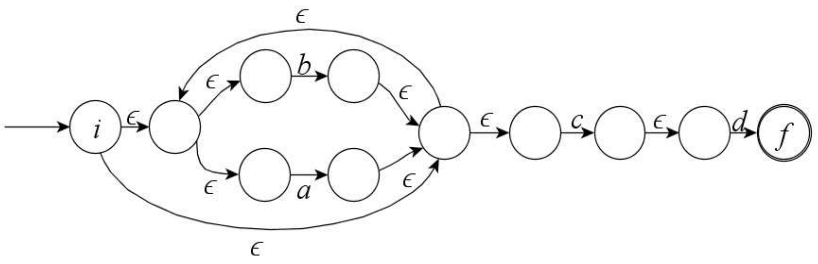


Figure 2.3: NFA for the regexp (a|b)*(cd) (Sidhu and Prasanna 2001).

A variation of the DFA and NFA also exists and is called a Hybrid-FA (Becchi and Crowley 2007b). The Hybrid-FA attempts to bring together the strengths of both DFAs and NFAs, and all the nodes likely to lead to state explosion whenever a DFA is created from an NFA are forced to retain an NFA encoding. The rest of the states are then transformed into DFA nodes.

DFA state explosion occurs when trying to convert DFAs into composite DFA from a given regexp. The regexp involved could be a single regexp with repeated wild cards (.*) and length

(constrained) restrictions, or regexps with multiple regexps combined together into a composite DFA. In the latter case, each of the combined regexp will have wild cards with length restrictions. The period (.) also called dot in the wild card represents a single character, while the character (*) placed after the dot implies that any number of character could be matched. The length restriction dictates the number of times a character or class of characters are repeated in regexp. Although in practice, if compiled in isolation, individual regexps with single wild cards mostly found in current rules do not lead to state explosions. However, the number of transitions could be affected, but not the number of states as observed by Becchi and Crowley (2007b).

Generally '...if a regexp matches exactly j arbitrary characters, [then] 2^j states are needed to handle the exact j requirement' (Yu et al. 2006). A DFA for a regexp of length k and a length restriction j could potentially have up to $k+2^j$ states in worst case as observed by Yu et al. (2006). A good example of such a pattern is: .*ab.{20}cd. For such a regexp, there could be at least: $k+2^{20} = k+1048576$ states in the DFA, which is composed of over a million states. This gets worse with every increase in the length restriction. The hybrid-FA exploits the fact that DFAs corresponding to the rules obtained from NIDS such as Snort show significant level of state transition redundancy. More so, such redundancies can easily be exploited by a lot of approaches (refer to Section 3.2.1, for more details). For the purpose of this thesis, focus is on the use of NFA at all times. This is because it is the choice of automata for implementing the approach in this thesis as earlier introduced in Chapter 1 and fully discussed in Chapter 5.

There is a need to effectively match against complex regexps that repeatedly occur in current Snort rules. However, rules such as the ones provided by Snort are integrated into an Intrusion Detection System (IDS). Section 2.4 discusses more on the classes of IDS and their uses, with particular focus on Snort IDS.

2.4 Network Intrusion Detection System

Whenever there is an attack on network-based computer systems, the first line of defence is the use of intrusion prevention. Preventive methods such as the use of passwords and biometrics (Zhang and Lee 2000, p. 2) which are implemented within techniques like: encryption and authentication are helpful but insufficient. The increased number of exploitable program bugs and errors, as well as technical and human weaknesses further compounds the problems faced by preventive measures. As such, intrusion detection has become the second line of defence necessary for protecting a computer network system. An intrusion detection system attempts to capture data from a given network traffic and then applies rules to it in order to identify anomalies present in the data packet. Intrusion detection system can be categorised as a network-based (NIDS) or host-based (HIDS) intrusion detection system (IDS) (Bolzoni and Etalle 2008; Rehman 2003; Zhang and Lee 2000). A third category is the distributed IDS as described by Singh et al. (2015, p. 1).

A NIDS runs at a network gateway and analyses the network traffic in order to detect unauthorised accesses and harmful contents flowing through the network. Also, the NIDS matches harmful contents to a known database of signatures, and generates an alert or logs the packet to a file or database. As such, Snort IDS is specifically designed to function as a NIDS. A HIDS acts as an agent that checks through the logged files of a system or application for suspicious activities. The HIDS is heavily dependent on the audit data provided by an operating system, which does the monitoring and evaluation of generated

events by the host programs or users. The distributed IDS collectively analyses events from many sources for traces of malicious activities (Singh et al. 2015, p. 1).

Snort IDS is categorised on the basis of the techniques used for its implementation, just like other known IDS. The two main categories of IDS techniques are: signature-based (SBS) or misuse detection, and anomaly-based (ABS) IDS (Bolzoni and Etalle 2008; Zhang and Lee 2000). The SBS system such as Snort (Sourcefire 2013; Roesch 1999) is mainly concerned with utilising pattern matching techniques for detecting intrusions. The intrusions are based on stored databases of the signatures belonging to known threats. The SBS then attempts to match these known signatures against the analysed incoming data packets in real time. However, an ABS is implemented differently from the SBS, because an ABS 'first builds a statistical model... [that describes] the normal network traffic,... [before flagging] off behaviours that significantly deviate from the model, as an attack' (Bolzoni and Etalle 2008, p. 2).

Accordingly, an SBS NIDS typically applies string matching only to those sections of the packets likely to contain the offending data. For instance, Snort NIDS is a packet sniffer and logger designed to act as a lightweight NIDS to perform content pattern matching. The Snort NIDS detects different types of attacks and probes including: buffer overflow, Denial-of-Service (DoS) and other malware. This makes the NIDS a popular solution. Also, Snort NIDS has a real-time capability and sends alert updates to syslog, and server message blocks (SMB) using separate alert files (Roesch 1999). Snort NIDS attempts to check from the beginning that a given packet header has a high likelihood of containing hostile data before performing the highly compute-intensive task of examining the packet payload in more detail. However, while this may be efficient in detecting hostile packets, malicious packets can occasionally be overlooked and consequently allowed into the network (Hutchings, Franklin and Carver 2002, p. 111).

For the purpose of this thesis, focus on the design is on the implementation of a framed SBS pattern matching problem. The framed problem is solved using the novel pattern matching approach introduced in Section 1.4. The design approach is vital because it detects malicious attacks that occasionally slip through a system such as Snort NIDS. This is achieved by independently carrying out the highly compute-intensive task of performing a deep packet inspection of each packet payload that is streaming through the network at high speed (Yang and Prasanna 2012; Singapura et al. 2015). The design can be comfortably placed physically between a network firewall and a home network.

2.4.1 Network Security Issues

Computer networks are vulnerable and could easily become insecure due to the various attacks it experiences on a daily basis. The attacks could compromise the network security in place, leaving it vulnerable to potential exploitation. These potential threats could be in the form of unauthorised accesses, spams, bugs, Denial-of-Service (DoS), malicious software and other related threats. There are four key threats which Aycock (2006, p. 1) identifies and refers to as the 'four horsemen of the electronic apocalypse' namely: spam, bugs, DoS, and malicious software. The following briefly explains each of them:

1. Spam: Is a term that is commonly used to refer to the heavily unsubscribed emails that Internet users receive almost on a daily basis, which Aycock (2006, pp. 1-124) describes as '... plagues [to] the mailboxes of Internet users worldwide'.
2. Bugs: These are program errors capable of stalling the execution of installed software. Such errors are hard to find, thereby posing a serious security flaw.

3. Denial-of-Service (DoS): A DoS attack typically slows down a network or computer system until it crashes. A DoS denies certified users fast access to authorised resources or services by simply flooding a machine with unsolicited requests.
4. Malicious software: Malicious software has dangerous intent and includes the following: viruses, worms, Trojan horses, and spyware and adware.
    a. Virus: A virus is a self-replicating program code, capable of infecting system files. Virus only spreads locally within a computer, infecting its system files. A windows-based virus is an example of a virus that lives on a Unix-based file server. The virus remains inactive until it is exported and executed on a Windows-based machine(s) and executed.
    b. Worm: A worm is a malicious program that shares common characteristics with a virus. But unlike viruses, worms spread from computer to computer without any human intervention. A worm can replicate itself on a system in thousands, and that can cause individual computers, networks and web servers to slow down or to stop responding completely. A worm such as the Blaster worm, tunnels through a system and allows remote access. The worm can also overload network resources with the amount of traffic it generates.
    c. Trojan horse: A Trojan horse in computing is a program which does not replicate or infect other files, unlike worms and viruses. It is a malware designed to behave like a genuine program but ends up secretly performing some extra tasks considered to be malicious. An executed Trojan can change desktop settings, damage and even delete system and user files. Some Trojans often called backdoors could even open doors for its attacker(s) to gain access into a system, thereby allowing confidential or personal information to be compromised.
    d. Spyware: A spyware is software that gathers information from unsuspecting computers and transfers it to the interested party. For instance, user names and passwords could be extracted from stored files on a computer, or recorded using what Aycock (2006) describes as a ‘keylogger’. The keylogger is described as the variation of a Trojan, which captures keystrokes only, without any vigorous trickery involved. Other information of interest include: email addresses and potential transaction details such as bank accounts and credit/debit card details.
    e. Adware: An adware bears resemblances with a spyware. The adware is another program that gathers information about the lifestyles of unsuspecting computer users without their consent. Such malware specifically targets market-focused advertisements or is ‘[used to] redirect a user’s web browser to certain websites in the hope of making a sale’ (Aycock 2006, p. 17).

Security weakness is another source of concern to most networks. Such weaknesses can be categorised into two main categories, namely: technical and human weaknesses.

1. Technical weakness: These are weaknesses that are mostly software related. A good example of a technical software weakness is buffer overflow. A Buffer overflow occurs when the limit of an array, often a buffer in a given code is exceeded. An attacker could easily exploit such a weakness in a given code, and a good example of a typical buffer overflow is stack smashing.

    A stack smashing attack has to do with the prior knowledge of where the buffer is kept in a stack, which happens to be a local variable in a given code. A little flaw easily occurs when a

bound is never placed on the input being read. As such, when the stack-allocated buffer begins to fill up from low to high memory, the attacker can persistently write over the top of the return address on the stack. Furthermore, if the attacker's code is a shellcode that is followed by the associated return address, then as the fill_buffer returns control, it will resume the execution to the location where the attacker specified it, and then run the shellcode (Aycock 2006).

2. Human weaknesses: These are weaknesses that are strictly attributed to humans. By forgetting to apply even the most basic security patches, users could potentially allow bugs into their networks or computer systems. Sometimes software with known vulnerabilities are installed or improperly configured. This could potentially leave holes for attackers to exploit. A good example is a fake email request. The email request could demand a user's attention to respond to a fake reward for some unsolicited lottery won. The attackers usually request the users to divulge their usernames and passwords in order to claim their prizes as observed by Aycock (2006).

The knowledge of the categories of security issues discussed in this section is needful. Once the type of intrusions are known and stored as signatures in any rules database, then rules such as the once found in the Snort rule can then be properly applied in real-time to check against incoming data packets.

Furthermore, the design in this thesis required a form of circuit description during synthesis (refer to Section 2.7 for more details on synthesis), simulation and implementation. To efficiently describe such circuits, a fast processing hardware platform is needed, and a good candidate is the FPGA. FPGAs are programmed using hardware description languages. A hardware description language is a specialised standard text-based computer language used to describe the behaviour and structure of a given system and circuit design (Xilinx 2012a). Section 2.5 discusses more on hardware description language, particularly the type used to describe the various regular expression matching engine (REMEs) circuits. The REMEs were introduced in Section 1.4 and 1.7. However, Section 5.3 and 5.4 discusses the thesis approach and its full design implementation.

2.5 Hardware Description Language

A Hardware Description Language (HDL) is a program language used to describe digital circuits. It is common practice to use both HDL and synthesis software to build digital hardware. However, synthesis software cannot automatically derive a physical hardware on its own simply because the HDL codes are devoid of syntax errors. In other words a poorly written HDL code description cannot be easily translated into highly optimised and efficient implementation. Although, synthesis software is capable of performing HDL transformation and localised optimisations (Chu 2006), there is still a need to properly understand the structure of a HDL. This is because generating complex hardware or non-synthesisable HDL descriptions can easily be avoided by having proper knowledge of structure of the HDL. A HDL typically describes the various concepts involving connections between circuit parts, the concurrent operations and the propagation delays and timings that occur. Each type of HDL is designed to provide a solution for a peculiar application. For instance, the HDL called Constructing Hardware in a Scala Embedded Language (Chisel) (Bachrach et al. 2012, p. 1217) is designed to simply provide modernised features of current programming languages. The language is capable of specifying low-level hardware blocks. The idea behind Chisel HDL is to make it extendable enough 'to capture...[important] high-level hardware design pattern' (Bachrach et al. 2012, p. 1217).

VHSIC Hardware Description Language (VHDL) was used as the HDL of choice to implement the design in this thesis. The choice of VHDL is purely based on familiarity and ease of use. While higher-level programming languages like Java is used to describe algorithms with sequential execution, VHDL is used to describe hardware that deal with more parallel execution. Mealy (2007) describes two primary purposes for a HDL like VHDL thus:

a. The modelling of digital circuits which is a description of something that presents a certain level of detail is possible.
b. Also, for a given circuit model, there is subsequent ease of simulation and/or testing of the circuit.

VHDL has a considerable advantage when efficiently describing complex logic as follows:

a. It describes a system's structure. Such as the decomposition of the system into subsystems and their relative interconnections.
b. It specifies the function of a system, using familiar programming language constructs.
c. It allows a system design to be simulated before being implemented and consequently manufactured.
d. VHDL provides an easy mechanism for producing the device-dependent version of a design, which is synthesised from a more abstract specification. The specification leads to decisions that cut down on the overall market time for the design.

The basic building blocks of VHDL are used in almost every design description. The description together with some redefined terms that have some meaning to the average designer is described by Perry (2002, pp. 2-3) as follows:

a. Entity: An entity is described as the most basic building block in a design and all designs in VHDL are expressed in form of entities. The entity defines the I/O ports, which are the I/O entry points into the entity (Xilinx 1999).
b. Architecture: Every entity to be simulated has at least one architecture description that describes the behaviour of the entity. One entity could contain multiple architectures, with each serving a different purpose like describing behaviour, and structure of the design etc.
c. Configuration: A configuration is described as a statement that secures a component instance to an entity-architecture pair. The statement describes which behaviour is used for each entity.
d. Package: A package is described as a collection of repeatedly used data types and the subprograms used in a given design. A package is like a tool box.
e. Driver: A driver is described as a signal source. A signal driven by more than one source will have multiple drivers if all the sources are active.
f. Bus: In VHDL, a bus is a unique signal type, which refers to a group of signals. In VHDL, the drivers of a bus can be switched off.
g. Attribute: An attribute is a predefined data. The data is usually devoted to VHDL objects. An example is the highest operating temperature of a device.
h. Generic: A generic term passes information to an entity. For instance, the number bytes to be read as input from a test bench could be passed into the entity as generic values. A test

bench is a HDL description that permits a designer to provide a documented and repeatable set of stimuli, which is portable across different simulators

i. Process: This is the primary unit of execution in VHDL. All the operations executed in a simulation of a VHDL description are split into single or numerous processes.

The VHDL circuit descriptions are were written for the target reprogrammable device referred to as Xilinx FPGA Virtex-6 device. The device is bundled with the XST VHDL synthesis tool (refer to Section 2.7 for more details on XST VHDL synthesis tool). Section 2.6 discusses more about the FPGA device.

2.6 Field Programmable Gate Array

A Field Programmable Gate Array (FPGA) is a general-purpose reprogrammable semiconductor silicon device. The performance and function density of FPGAs have improved over the last two decades. The improvements were made possible due to the continuous decrease of device sizes (Trefzer 2015, p. 1). Also, the fast FPGA clock rate is capable of enhancing design performance (Brodie, Taylor and Cytron 2006). The FPGA device is centred on a matrix of Control Logic Blocks (CLBs) which are connected through programmable interconnects (Xilinx 2014). The FPGA is designed for reconfiguration by users, after it is manufactured. The reconfiguration feature of the FPGA is the reason why it is called a field-programmable device. An HDL such as VHDL is usually used to program and specify the FPGA configurations. Currently FPGAs easily push the 500MHz performance barrier (Xilinx 2014), and provide software flexibility in terms of programming. The benefit of hardware performance such as parallelism, which allows for thousands of parallel executions, is also an added advantage of the FPGA (Moussalli 2014, p. 1).

Timing verification for FPGA designs is required to check if the desired output responses are produced within stipulated timing constraints. For instance, the time it takes to setup and the time it takes to hold input signals need to be properly defined for correct timing computation. Black-Schaffer (2003, p. 4) defines a setup time as the ‘amount of time the synchronous input (D)... [of a D-type flip-flop, such as a D-flip flop (DFF)] must be stable before the active (rising) edge of the clock’. Also, Black-Schaffer (2003, p. 4) defines a hold time as the ‘as the amount of time the synchronous input (D) of the DFF must be stable after the active edge of the clock’. Verifying the correct timing of valid data within the setup and hold time is a big challenge for FPGAs. The timing specification is important in determining the overall speed and consequently the design throughput. The throughput is usually limited by the operational clock frequency or hardware clock. As such optimising the operational clock frequency is necessary for achieving optimal performance. This is difficult to achieve because the performance of a design is highly dependent on correct timing constraints (Chu 2006).

Nevertheless, FPGAs can be partially reconfigured. Partial reconfiguration also allows FPGAs to reconfigure selected areas at any time after its initial configuration. Partial reconfiguration is also an important feature that is used in communication devices. For instance, a communication device may control multiple connections, some of which require encryption. As such, it would be useful to be able to load different encryption cores without bringing the whole controller down when it is running. However, partial reconfiguration is not supported by all FPGA devices. From the functionality of any design, partial reconfiguration can be divided into two groups:

i. Dynamic partial reconfiguration: Also known as an active partial reconfiguration (Lysaght et al. 2006, p. 1; Lie and Feng-yan 2009, p. 445). Such reconfiguration permits changes to be made to

a part of the FPGA, while the rest of it is still running. A description of special regions called partial reconfigurable regions (PPR) describes the area of the FPGA that is reserved for implementing all tasks in one dynamically reconfigurable subset. Such a subset was utilised by Wang et al. (2010, p. 214) for implementing a character class with constrained repetition called a CCR-based scanner approach. In the approach by Wang et al. (2014), partial reconfigurable modules (PRM) were used to describe the implementation of a single dynamic task that was later mapped into a PPR.

ii. Static partial reconfiguration: During the process of static partial reconfiguration, the device is not active. While the partial data is sent into the FPGA, the rest of the device is stopped in the shutdown mode, and only activated after the configuration is completed (Lie and Feng-yan 2009, p. 445).

Reconfigurable approaches take advantage of the nature of specific rules, like those found in the current popular Snort rules (Snort 2013). Any change to the rules will require the re-generation of a new circuit usually written in HDL such as VHDL. The HDL file is then recompiled, re-synthesised and re-implemented in the FPGA (Tan and Sherwood 2005). A good example of such reconfigurable design is the one deployed by Clark and Schimmel (2004) using a multi-character decoding approach. The design was based entirely on similar reconfigurable approach.

Full reconfiguration is required, especially by approaches that constantly either try to save more resources or improve performance. The performance could be measured in terms of design density. The density 'is a measure of an implementation's capacity per unit area' (Brodie, Taylor and Cytron 2006). Density can be measured in die area (measured in mm^2) or device specific resources such as LUTs in an FPGA.

Furthermore, LUTs use a simple array indexing operation to retrieve data from memory. The retrieval process is performed without incurring any costly runtime computation or I/O operation. This is because retrieving data from memory is often faster than carrying out some complex routine with more procedures to achieve the same. In digital logic, an n-bit LUT with n = 2, 3, 4, 5, 6 and 7 can be implemented with a multiplexer. The select lines of the multiplexer are the inputs of the LUT. The approach in this thesis measured the throughput efficiency based on the ratio of the number of LUTs (Yang, Jiang and Prasanna, 2008; Clark and Schimmel 2004) utilised by each of states of the automata. The resulting value is then multiplied by the overall throughput of the design. Section 6.2.3 discusses more on the throughput efficiency computation process.

The CLBs contained in a Xilinx FPGA Virtex-6 device (Xilinx 2012a, p. 7) are the main logic resources used for implementing both sequential and combinatorial circuits on FPGAs. The CLBs can be configured to provide functionality as simple as that of an inverter or as complex as that of a microprocessor. CLBs can also be used to implement different combinations of combinatorial and sequential logic functions. Such functions include: combinational gates like basic NAND gates or XOR gates. Other functions include: n-input LUTs with n = 2, 3, 4, 5, 6 and 7, multiplexing, and wide fan AND-OR structures. Each CLB contains two components called slices. The slices are connected to a switch matrix for accessing the general routing matrix as shown in Figure 2.4. Each slice (Xilinx 2012a) component contains 'four LUTs, eight storage elements (Flip-Flops or FFs), wide-function multiplexers, and carry logic'. The routing matrix is a flexible programmable unit used to connect the CLB logic blocks

with each other. The connection is such that each CLB is connected to a switch matrix for access to the general-routing resources, which runs vertically and horizontally between the CLB rows and columns. A similar switch matrix connects other resources, such as the digital signal processing (DSP) slices and block RAM resources (Xilinx 2014, p. 72). Figure 2.4 shows how a typical Xilinx FPGA Virtex-6 device CLB looks like (Xilinx 2012a, pp. 7-11). The first slice in the CLB labelled slice (0), is positioned at the bottom of the CLB, while the second slice labelled slice (1) is positioned at the top of the CLB.

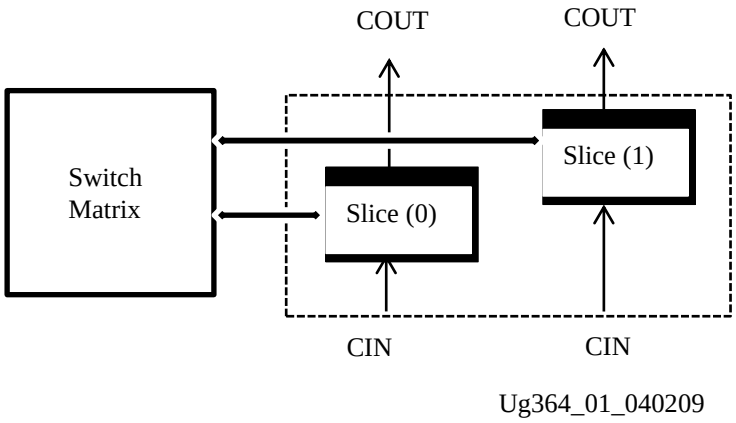


Figure 2.4: Arrangement of slices within the CLB (Xilinx 2012a), p. 7).

The CLB slices have no direct connections between them. Each slice is organised as a column, with each having an independent carry chain. The LUTs within a slice can be implemented as a synchronous RAM, which is referred to as a distributed RAM. An n-bit LUT, with n = 2, 3, 4, 5, 6 and 7 can encode any n-input Boolean function by modelling such functions as truth tables. The function generators in a Xilinx FPGA Virtex-6 device are implemented as 6-input LUTs (Xilinx 2012a). A distributed RAM is used for storage, while a 32-bit wide shift register is used for shifting data.

The distributed RAM and the shift register are available only in a particular type of slice called SLICEM as seen in Figure 2.6. The counterpart of the SliceM is the SliceL. The upper-case letter "L" at the end of SliceL stands for 'logic', while the upper-case letter "M" at the end of SliceM stands for 'memory'. The SliceL does not have a distributed RAM or shift register, which makes it different from the SliceM. However, the LUTs in a given SliceM as shown in Figure 2.6 can be made to function as synchronous RAM which is simply a distributed RAM. The SliceL LUTs can only function as a random combinational logic. Furthermore, the distributed RAM is a 256-bit RAM (4x64-bits) component and the shift register is a 128-bit (4x32-bits) register. The distributed RAM modules perform synchronous (write) function. A 'synchronous read can be implemented with a storage element... [such as a] flip-flop' (Xilinx 2012a). The FFs are eight in number within any given slice, and are configured as positive edge-triggered D-type FFs. Furthermore, the SR and CE signals are active High, where a higher voltage represents the binary digit of 1, used to assert the state of a logical condition.

The LUTs in a SLICEM can be combined in different ways to form a larger storage for data (Chaves et al. 2008). Every slice contains three multiplexers namely: F7AMUX, F7BMUX, and F8MUX. A multiplexer is used to 'combine up to four LUTs in order to provide any function of seven or eight inputs within the slice' (Xilinx 2012a). The F7AMUX and F7BMUX are used to generate inputs from any of the four LUTs in a given slice. The F8AMUX is used for joining all the four LUTs in order to

generate eight input function as shown in Figure 2.6. Figure 2.5 shows how the XST VHDL synthesis tool designate slices. An "X" followed by a number identifies the position of each slice in a pair as well as the column position of the slice. The "X" number counts slices starting from the bottom in sequence 0, 1 (the first CLB column); 2, 3 (the second CLB column). A "Y" followed by a number identifies a row of slices. The number remains the same within a CLB, but counts up in sequence from one CLB row to the next CLB row, starting from the bottom (Xilinx 2012a, p. 6).

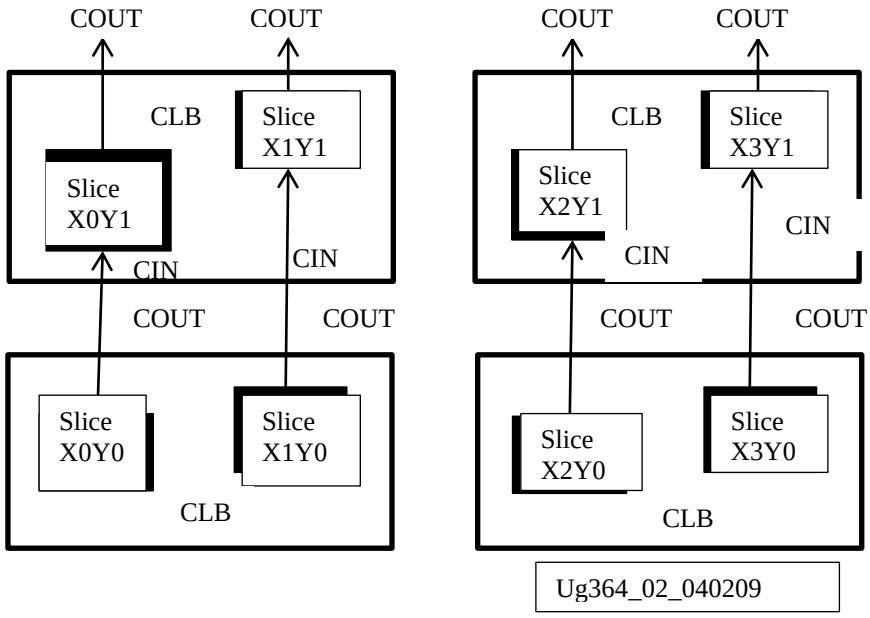


Figure 2.5: Row and Column relationship between CLBs and Slices (Xilinx 2012a, p. 8).

From Figure 2.6, the section labelled "LUT RAMs" implements 6-input LUTs, and each of the four LUTs has six distinct inputs labelled A1-A6 and two distinct outputs 05 and 06. Each of the LUTs perform 6-input Boolean functions arbitrarily. The 06 outputs are used whenever a 6-input function is applied. Irrespective of which function is implemented, the propagation delay through a LUT is independent, and the signals leaving each LUT can be through the 06 output or any the multiplexers (MUXs) outputs for the 05 output. The generated signals then enter the labelled "Carry-logic" in Figure 2.6, belonging to the 05 output. Also the same signal is passed to select lines of the carry-logic MUX belonging to the 06. The same signal is also passed to the D input of the DFFs or the F7AMUX/F7BMUX from the 06 output (Xilinx 2012a).

The DFFs or level-sensitive latches labelled as "FFs/Latches" in Figure 2.6 can be driven directly by a LUT output through the MUXs of each of the four LUTs or simply by the slice inputs which bypass the LUTS through the inputs namely: AX, BX, CX and DX. The latch is apparent whenever the clock signal is low. In the Xilinx FPGA Virtex-6 devices, only the last four out of the eight DFFs can be configured as edge-triggered DFFs. This means that whenever the first four DFFs are configured as latches, the four DFFs cannot be functional. However, the eight DFFs all have common control signal clocks (CLK), clock enable (CE) and set/reset (SR). When a DFF in a slice has set/reset (SR) or clock enable (CE), the other DFFs in the slice will also have SR or CE enabled by the common signal clock.

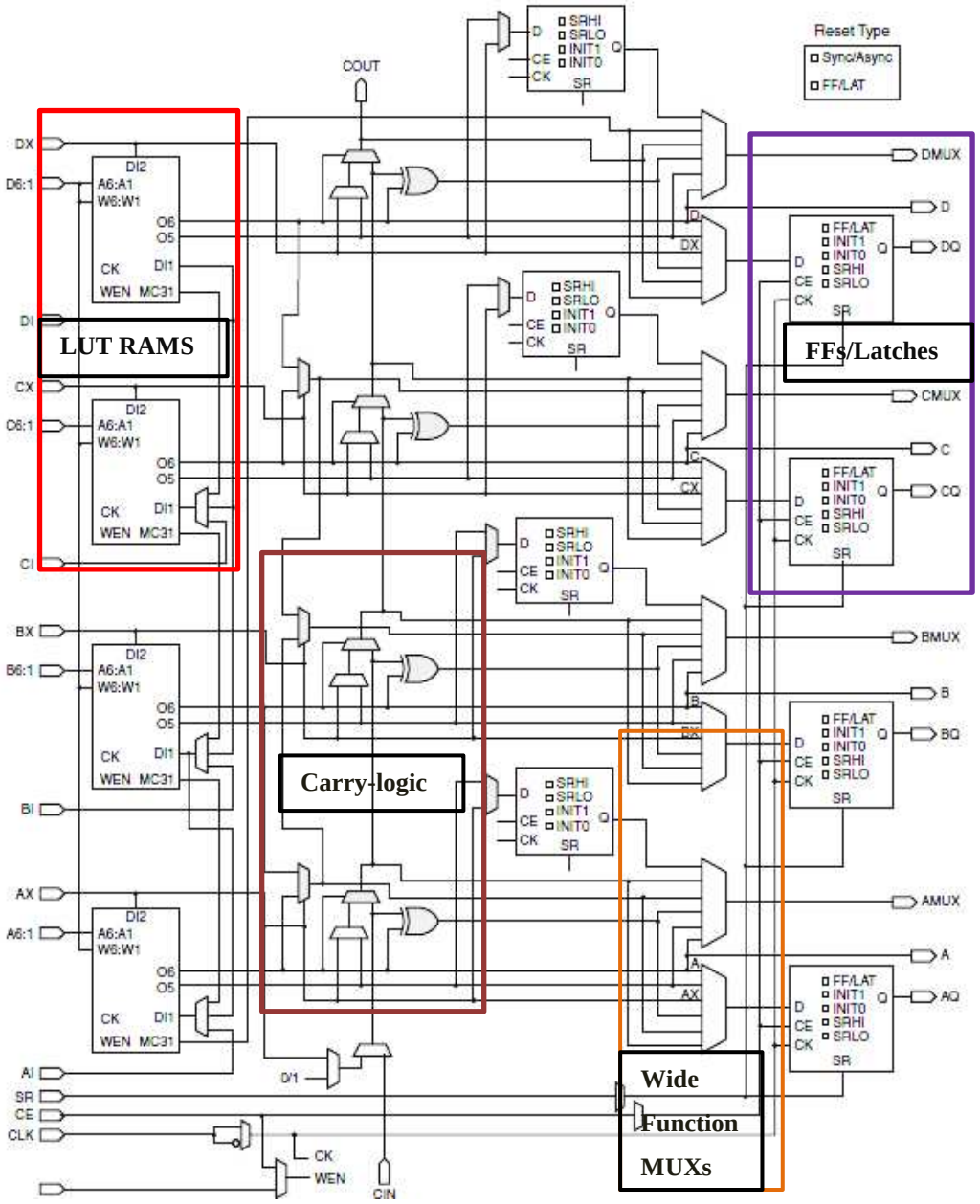


Figure 2.6: Diagram of a SLICEM (Xilinx 2012a. p. 9).

The FPGA arena (Mitra, Najjar and Bhuyan 2007) has witnessed a rapid acceptance and increase in speed and silicon logic size over the years. The latest devices support multi-gigabit throughput interfaces to the host processor. The FPGA is a preferred choice for many applications that use regexps. Such applications include: DNA sequencing, compilers, spam filters, and data mining applications, to mention but a few. The applications usually require periodic updates. Furthermore, the flexibility of the FPGA leverages the implementation of highly optimised parallel logic circuits, which support a multitude of REMEs.

2.7 Synthesis Process

Synthesis involves the process of translating HDL codes into a netlist. The netlist (Wain et al. 2006, p. 5) is the textual description of a circuit diagram or schematic. Verification is performed at the early stage of a design to determine if the design works according to specification and performance objective. The two main aspects of verification are ‘functionality and performance’ (Chu 2006, p. 14). Functional

verification ascertains whether the right output responses are generated by a given design. Performance on the other hand, is based on the timing constraints. As such, timing verification is used to check if the desired responses are produced within the stipulated time constraint. At different levels of design abstraction and phases, verification is performed. The design abstractions are: High-level, Register-Transfer level (RTL or RT-level), and Gate-level synthesis. Technology mapping is the fourth abstraction (Chu 2006, p. 14).

High-level synthesis converts an algorithm specified unambiguously in terms of register transfer operations into an RTL description. RTL synthesis develops the structural implementation by utilising RTL components. The implementation comes after analysing the RTL behavioural description. A number of utilised components are often reduced by carrying out a partial degree of optimisation during the RTL synthesis. RTL data representation can be abstract, especially when signals are mostly assembled together and understood as a special category of data types. Such a data type can be an unsigned integer or system state. Furthermore, with RTL abstraction, broad expressions are used to specify the functional operation and data routing at the level of the behavioural description, A comprehensive FSM is then used to describe a system that is designed using RT methodology. The use of a clock signal in RTL description is a main feature found in storage components such as shift registers.

Gate-level components such as AND-gates are utilised in structural implementation during Gate-level synthesis. Hierarchical optimisations are performed to minimise the circuit size or meet the various design timing constraints. Lastly, during the technology mapping process, each device technology such as the Xilinx FPGA Virtex-6 device (Xilinx 2012a) consists of predefined set of primitive gate-level components. The components are packaged within the generic logic cells of the FPGA device. As such the generic components need to be mapped specifically into the respective cells of the selected technology during gate-level circuit implementation (Chu 2006, p. 13-15). Technology mapping is the process of gate-level circuit transformation and it is technology dependent. The process constitutes the final synthesis step.

During the process of verification at the various levels of abstractions, commonly used methods such as simulation are used for building and executing the system model using some test patterns. Afterwards, the output responses are then analysed and properly examined for correctness. It should be noted that simulation does not guarantee that selected stimuli can be used to verify the correctness of the whole design. Furthermore, while simulation can be used to identify major design flaws, it cannot guarantee the absence of errors. It becomes even more difficult to simulate low-level models consisting of millions of components using computers that perform sequential computations. This is because hardware operations are usually concurrent and parallel in nature (Chu 2006, p. 15).

However, a behavioural simulation is performed using a test bench which provides the necessary stimuli. A test bench can be as simple as a file with clock and input data or a complicated file containing an error checking capability. It could also be an input and output file that has the ability to perform conditional testing. Timing analysis can also be applied to analyse the circuit structure, determine the various input-output paths. For instance, timing analysis can be used to compute the propagation delays of the paths adjust the timing parameters accordingly. Such timing parameters could be worst-case propagation delays and maximal clock frequency.

The XST VHDL synthesis tool used for implementation in this thesis, takes the description of a design in a HDL file discussed in Section 2.5, and converts the file to a synthesised netlist. This is achieved by first carrying out functional (or RTL) simulation of the design description. The generated netlist file is called a native generic circuit (NGC). The NGC file contains both the logical design data and the associated design constraints. The NGC file takes the place of both the Electronic Data Interchange Format (EDIF) and netlist constraints file (NCF) files (Xilinx 2008a). The NGC files are either generated by the synthesis tools, schematic editors, or other design entry mechanism. The NGC file is then translated into a binary file format. The components and connections defined in the NGC file then mapped to the Control Logic Blocks (CLBs) (Xilinx 2008b). After the mapping process is completed, the design is then placed and routed to fit onto a target FPGA (Wain et al. 2006, p. 5). To establish the completeness of the design, a second post place and route simulation is performed to ensure that the design is been properly placed and routed. The application of compiled tools such as SmartOpt (Jang et al. 2009, p. 239) integrated into the XST, also allows more complex FPGA architectures to be further optimised and mapped to simpler netlist models.

The synthesised netlist, which represents a logical view of the design, is processed during the design implementation (Xilinx 2008b) phases. The phases can be summarised thus:

a. Translate: This is when the netlist and constraints are merged into an NGC file.
b. Map: It is at this point that the components and connections defined by the NGD (logical design) file are mapped to FPGA components such as: CLBs and input/output blocks (IOBs). The output file is a native circuit description (NCD) file, which physically represents the design mapped to the various FPGA components such as shift registers, multiplexers, FFs and so on. The NCD file is then used by the programming file generator referred to as BitGen, to create a configuration bits file or bitstream (Xilinx 2010, pp. 136-137). The bitstream is used to program the target FPGA.
c. Place and route: It is at this stage that the mapped NCD file is taken and used to place and route the design, before finally producing a final NCD file. The final NCD file is then used as the input to generate the configuration bits file for programming the target FPGA.

Finally, the generated configuration bits file (Xilinx 2010, pp. 136-137) contains all the information required to program the target FPGA logic circuits. This is achieved by loading and executing the file in the target FPGA device. The final verification and debugging of the design is done by using a special tool called the Xilinx Chipscope (Wain et al. 2006, p. 5). This is done while the design is actively running on the target FPGA. A good example of an FPGA device with XST capability is the Xilinx ISE FPGA Project Navigator, version 14.4 design suit.

2.8 Chapter Summary

This chapter presented a general background leading to the understanding of regexps and pattern matching. An overview and brief description of intrusion detection and the various types of security issues were also discussed. The chapter briefly introduced what constitutes the FPGA hardware platform used for designing various design REMEs. XST was introduced in order to have some basic knowledge of what transpires during the design synthesis process and implementation. Chapter 3 reviews some of the various approaches that are most related to the approach implemented in this thesis.

3. Approaches to Regular Expression Pattern Matching

This chapter describes some important approaches used in designing REMEs namely: software and FPGA based approaches. The approaches are closely examined and later analysed in Chapter 4. The analysis gives a clearer understanding of how the various related approaches work in comparison to each another. The discussions on the various FPGA-based classification approaches builds up to the approach in this thesis, as discussed towards the end of the chapter. Chapter 5 discusses the full implementation of the design approach.

3.1 Introduction

The focus of this thesis is based on FPGA-based approaches, specifically the ones that deal with various forms of classification techniques. Nevertheless, Section 3.2.1 highlights some important general-purpose processor-based platforms. Specifically, approaches that deal with the equivalence and non-equivalence of states on a given FA are discussed. Hardware-based approaches exploit a high degree of parallelism and memory bandwidth, while creating compact automata accessible on on-chip memories. The amount of memory required for the implementations of such hardware-based architectures is considered to be high, often requiring many small on-chip memories. Memory bandwidth can also be a serious limiting factor for such systems. As such, the task of exploiting different techniques to reduce the number of required off-chip memory accesses is vital. The most common competing requirements are: design speed, memory, throughput and throughput efficiency. Most of the approaches described in this chapter sacrifice increased design throughput for lower memory utilisation. Other approaches try to increase the design throughput but end up with poor memory utilisation and throughput efficiency. The competing requirements have always been a trade-off, but the approach in this thesis has developed a way to balance such requirements. Section 3.2 reviews some of the originally developed string and trivial regexp matching approaches

3.2 Pattern Matching Design and Implementations

3.2.1 General-purpose and Processor-based Approaches

a. String Matching Approaches

An early compiler-search approach that locates specific character strings embedded in a character text was implemented by Thompson (1968, pp. 419-420). The compiler-search algorithm was incorporated as a context search compiler, applicable within a time-sharing text editor. The compiler was used in other

applications such as a symbol table search routine in an assembler. An IBM 7094 program was generated by the compiler as an object language from an acceptable regexp, considered as the source language. Consequently, the object language then generates a signal whenever an embedded string in the text matches a given regexp. The text serves as input to the object language (Thompson 1968, p. 421).

A sub-linear search algorithm that searches for the characters of a pattern in a given text string precisely once was implemented by Boyer and Moore (1977, pp. 762-763). Sub-linearity in this case refers to the way the expected number of inspected characters in a given string decreases as the pattern length increases. The search is made in worst case time of O(nm) and average case time of O(n/m), where m is the pattern length, and n is the text length. With information gained by beginning the search at the tail end of the pattern, the algorithm is able to make significant leaps through the given text being searched.

In another algorithm implemented by Knuth, Morris and Pratt (1977, pp. 323-328), all occurrences of a pattern in a given text could be found in average case linear time of O(n) and worst case linear time of O(m+ n), where m is the pattern length, and n is the text length. This happens without backing up the input text the way traditional approaches do. The process of backing up the input text involves some complicated buffering operations that are frequently involved in the matching process. The algorithm that only requires O(m) internal memory locations 'if the text is [to be] read from an external file, with only O(log m) units of time...[spent] between consecutive single-character inputs' was described by Knuth, Morris and Pratt (1977, pp. 323).

A finite state string pattern matching machine was also implemented by Aho and Corasick (1975, p. 333). The machine searches within a given text string to find a match for a finite sequence of strings called a keyword. The algorithm construction time for the state machine is proportional to the keyword lengths summed together. The main objective of the approach is to implement a library bibliographic search program. The program enables the bibliographer to easily locate in a given citation index, all the titles that satisfy some Boolean function of keywords and phrases.

However, the running time and number of required character comparisons in string matching is greatly influenced by the type of patterns, the data size and pattern lengths. The difficulty faced by such string matching approaches lie in the complexity of the current network attack signatures. Matching against such signatures often requires highly computationally intensive processes that consume a lot of the Central Processing Unit (CPU) processing time. As such, with a careful design, the expressive nature of regexps matching designs can be utilised to match against the complex attack signatures found in most current NIDS rules. Section 3.2.1b discusses some regexp matching designs, and highlights the main design structures and the processes involved.

b. Regexp Matching Approaches

i. Pseudo-equivalent State Merging Technique

A technique that works by first compiling the string patterns into a FSM was described by Lin, Tai and Chang (2007, p. 11). The FSM matches any substring of input strings contained in any string pattern. It was observed that there is a high performance cost involved, and that the power consumed by the memory architecture relate directly to the size of the memory in use. As a result, the idea of reducing the table size of the required memory became necessary.

Lin, Tai and Chang (2007) further noted that 'two states are equivalent if and only if their next states are equivalent'. Also, two or more string patterns could have similar common substrings, whenever the patterns are converted into a FSM. Although the commonality between the substrings means that their respective states will have similar transitions, having similar transitions alone does not make them equivalent states. Furthermore, because such non-equivalent states cannot be merged directly without creating the problem of false positive matches, a reduction to the FSM's states and transitions becomes difficult. As a result, 'Two states are defined as pseudo-equivalent states if they have identical inputs, failure transitions and outputs' (Lin, Tai and Chang 2007).

A state-traversal mechanism was then introduced, and it involved reviewing the classical Aho-Corasick (AC) algorithm (Aho and Corasick 1975). The aim was to exploit the algorithm's potential for reducing the number of state traversals on a FSM. As such, given a current state and an input character, the AC machine is tasked with determining whether or not the input character can trigger a valid transition. If the AC machine fails to trigger a valid transition, it is forced to jump to the next state where the failure transition points to. The same character is then considered repeatedly until the character causes a valid transition. The FSM formed by merging its pseudo-equivalent states is called merge_FSM as shown in Figure 3.1.

The FSM is formed from the AC machine based on the pattern: (a) "bcdf", and (b) "pcdg". The dotted lines signify failure transitions, while the thin lines signify valid transitions. The new merge_FSM significantly lowers the number of states and transitions realised, leading to lower memory requirements. But, the merging of pseudo-equivalent states can result in creating a FSM with functional error. A functional error occurs when a final state is erroneously reached by a pattern say P_2, instead of a pattern say P_1. Such an error in matching leads to a false positive match. For instance, given the input string "pcdf", the merge_FSM in Figure 3.1 erroneously leads to an accepting state 4, and wrongly reports a match as observed by Lin, Tai and Chang (2007). The reason for the false positive match is because neither of the patterns "bcdf" and "pcdg" was supposed to match the substring "pcdf". Ordinarily, a traditional FSM would have simply triggered a failed transition and returned back to its initial state 0 instead.

To avoid having such false positive matches for instance, a mechanism was implemented that understood what distinguishes between the pseudo-equivalent state 2 and state 6 in the merged state 26 of the merge_FSM of Figure 3.1. The mechanism was such that if the predecessor state is 1, when reaching state 26, then it could be ascertained that in the original FSM it was state 2 that would have been reached. State 6 would have been reached in the original FSM, if state 5 was the predecessor state. To realise such a solution, A state traversal process capable of memorising the precedent paths entering into the merged states was required. The idea behind the implementation of such a traversal process is to ensure that all the merged states in the merge_FSM can easily be differentiated. The design reuses those memory locations storing zero vectors in form of {00}, as well as the non-zero match vectors to keep useful path information called 'pathVec' (Lin, Tai and Chang 2007).

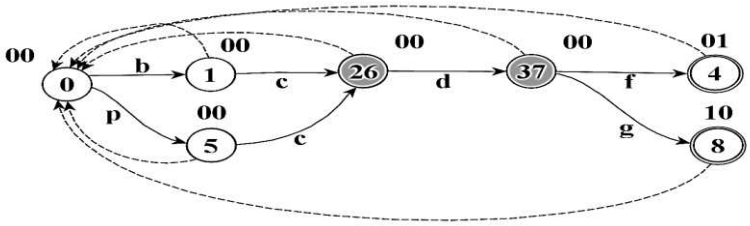


Figure 3.1: Merge_FSM formed by merging non-equivalent states (Lin, Tai and Chang 2007).

The pathVec of a given state say state 1 in Figure 3.2 is expressed as: $\{P_2P_1\} = \{01\}$, where P_2 represents pattern "pcdg" and P_1 represents pattern "bcdf". The least significant bit (lsb) bit value 1 represents P_1, while most significant bit (msb) bit value 0 represents P_2 in the bit vector {01}. In order to match the string "bcdf", the first bit value in the pathVec of the each state needs to be set to 1 for the given path 0->1->26->37->4 of the pattern. The second bit is set to 1 also for each state in the pathVec of the given path 0->5->26->37->8, which matches the string "pcdg" as seen in Figure 3.2. Lastly, an additional bit called the 'ifFinal' is introduced to each state, to indicate if a final state can be reached. Each state then stores the pathVec and ifFinal bit value in the format: 'pathVec_ifFinal' as shown in Figure 3.2 and Figure 3.3. For instance, from Figure 3.3 it can be seen that the accepting state 6 has a 'pathVec_ifFinal' value 001_1, which shows that the pattern "abcdef" has been matched.

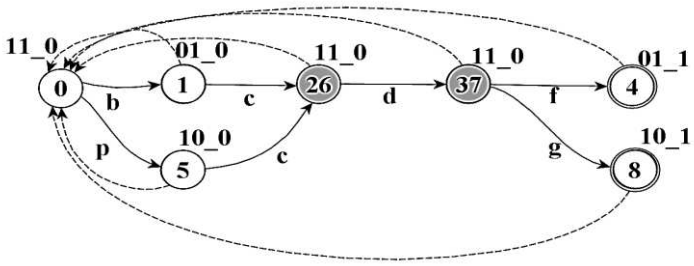


Figure 3.2: New state diagram of merge_FSM from Figure 3.1 (Lin, Tai and Chang 2007).

An additional register called 'preReg' was also created and used for tracing the precedent pathVec in each state. The register has a width equal to that of the pathVec. Each of the bits in the preReg represents a string pattern, and is updated in each state by performing a bitwise AND operation between the pathVec of the next state and the current preReg value. With the trace information, it is possible to keep track of the precedent path entering the newly merged state, making it possible to differentiate all merged states.

The process of constructing a state traversal machine starts with the creation of states and valid transitions. It is then closely followed by the construction of the pathVec and ifFinal starting in the first step and completing in the second step (Lin, Tai and Chang 2007). Given patterns: "abcdef", "apcdeg" and "awcdeh", the construction process starts by adding the first pattern to the directed graph. Afterwards, similar edges are reused between the patterns that were created as a result of sharing a common labelled character, such as the character a at the start of all the patterns as shown in Figure 3.3. Afterwards, more patterns are subsequently added to the graph incrementally to create a bigger graph of the three patterns as shown in Figure 3.3.

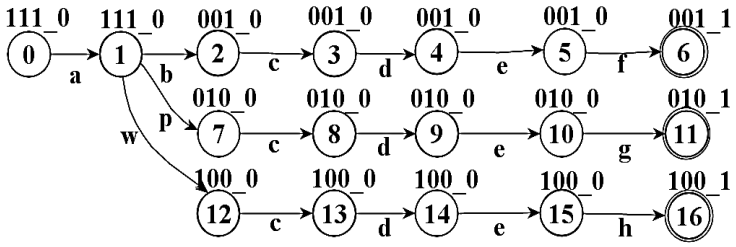


Figure 3.3: Construction of pathVec and ifFinal (Lin, Tai and Chang 2007).

For the construction of the state diagram belonging to the state traversal machine, the pseudo-equivalent states: 3, 8, and 13; 4, 9, and 14; 5, 10 and 15 are merged together into state 3, 4 and 5 as shown in Figure 3.4. The merging process is based on their common input characters of c, d and e respectively. The pathVec of state 3 is altered to be $\{P_3P_2P_1\} = \{001\}||\{010\}||\{100\} = \{111\}$, by performing a union operation on the pathVec of state 3, 8 and 13 of Figure 3.3. A similar operation is performed for state 4 and 5 to give {111} as well. The final state diagram as shown in Figure 3.4 eliminates 6 states from the original AC machine shown in Figure 3.3.

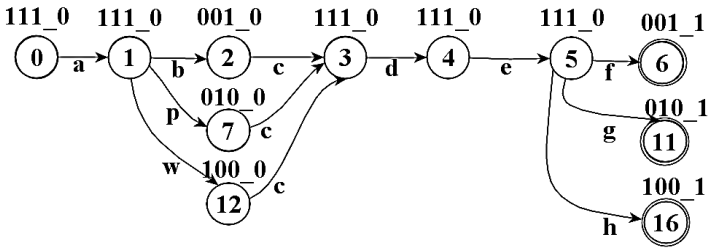


Figure 3.4: State diagram of the state traversal machine created from Figure 3.3 (Lin, Tai and Chang 2007).

Furthermore, an issue identified as a cycle (loop) problem (Lin, Tai and Chang 2007) could arise during the process of constructing a state traversal machine. The problem needed to be efficiently tackled or avoided completely. Cycle problems are created when merging multiple sections of pseudo-equivalent states in a merge_FSM. The cycle generated could lead to false positive matching results. Lin, Tai and Chang (2007) observed that for a cycle problem to be prevented from occurring during the creation of a merge_FSM, only pseudo-equivalent states that are not likely to form a cycle should be merged. Such pseudo-equivalent states must not be part of any disorder sections of the AC state machine before they are merged.

Sections are considered to be disordered if they have the potential to create a cycle in the process of merging their pseudo-equivalent states. The states involved must belong to the separate string patterns on the AC machine. For instance, the two patterns “abcdef” and “wdebcg”, which were used to construct the AC machine shown in Figure 3.5, caused a loop to exist by merging sections of the disordered pseudo-equivalent states. The states involved were state 4 and 7, as well as state 3 and 8 as shown in Figure 3.5. The new merge-FSM of Figure 3.6 created from the AC machine of Figure 3.5 now has a loop transition from state 5 to state 2 on input of character ‘b’. The loop causes the input string “abcdebcdef” for instance to be mistaken as a match for the pattern “abcdef” as shown in Figure 3.6. Such a mismatch is said to be a false positive match.

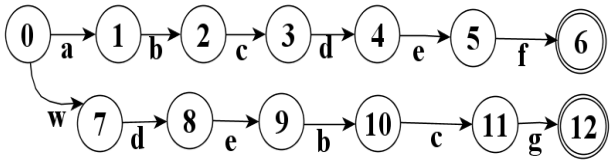


Figure 3.5: AC state machine for the two patterns “abcdef” and “wdebcg” (Lin, Tai and Chang 2007).

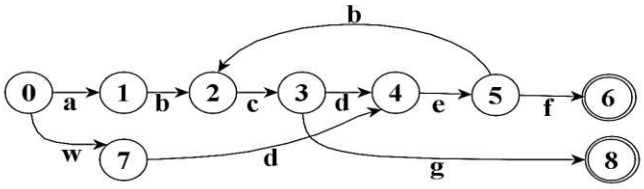


Figure 3.6: Merging two disorder sections of pseudo-equivalent states of an AC machine (Lin, Tai and Chang 2007).

In summary, adding multiple patterns to the directed graph could potentially lead to the creation of cycle problems. The cycles limit the amount of pseudo-equivalent states to be merged. The more the numbers of such patterns that cannot be merged, the more the limitation of such an approach, especially in terms of further memory reduction. As such one can conclude that the approach is not too memory efficient for multiple pattern matching applications. However, a proposed solution for resolving such cycle problems that can enable more patterns to merge together can be found in Section 7.2.1b.

ii. Delayed input DFA

Several packet content inspection engines have migrated from string matching to regexps matching, such as Snort (Roesch 1999) (refer to Section 2.3 for more on Snort NIDS) and Bro (Bro 2013). Bro (2013) is a powerful framework for network analysis, whose domain specific language allows for site-specific monitoring policies among other things. Cisco systems (Cisco 2013) have also integrated regexp-based content inspection capabilities into their Internetworking Operating System (written as iOS for short) among others.

An approach that performs a deep packet inspection was described by Kumar et al. (2006), called a Delayed Input DFA (D^2FA). The approach involves the scanning of every single byte of a packet payload with the aim of identifying predefined set of matching patterns. The memory architecture was created to utilise multiple on-chip memories. The design also ensures that each memory is uniformly applied and accessed over a ‘short duration, thus effectively distributing the load and enabling high throughput’ (Kumar et al. 2006).

The D^2FA is able to reduce memory requirement in comparison to most conventional DFAs. The approach incrementally replaces the several transitions from the DFA with a single default transition. A default transition can be described as an ‘unlabelled outgoing transition [from any given state]’ (Kumar et al. 2006) in the D^2FA. The thick line transition in the D^2FA (which is the figure to right hand side of Figure 3.7) represents the various default transitions, while the thin lines are transitions on character inputs. The D^2FA approach is based upon the observation that a number of groups of states on a traditional DFA (which is the figure to the left hand side of Figure 3.7) have identical outgoing transitions (also referred to as edges). The duplicate information about the identical outgoing transitions is then

exploited with the aim of reducing the needed memory requirements. A good illustration is presented in the automata as shown in Figure 3.7. Figure 3.7 shows the effect of the default transition on the automata, based on the visible reduction made in relation to the number of transitions. A failure to transit on any state of the DFA, leads to a backward transition to the previous state. Also two or more states share a common transition character as they transit to the same state. Kumar et al. (2006) considered such commonalities as a redundancy, which exists in the transitions. As a result, by introducing a default transition back to the initial state of the DFA, every regexp becomes an independent entity that leads to a unique final (accepting) state. Also by introducing the default transition, this will ensure that cycles only exist within each of the three regexps matched on the automata as shown in the D^2FA of Figure 3.7.

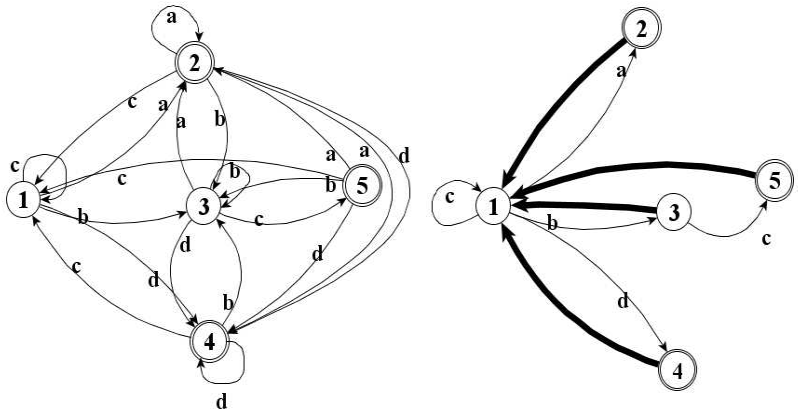


Figure 3.7: Examples of automata which recognise the regexps: a+, b+c and c*d+ (Kumar et al. 2006, p. 342).

In order to guarantee that the D^2FA meets the throughput objective of the design, a restriction was placed on the length of the longest default [transition] path. The default path is a path that comprises only default transitions. This helped to positively change the current version of the design. The aim was to find the least equivalent D^2FA which satisfies a specified bound on the default path length. The worst-case performance of a D^2FA based on the length of its longest default path was bound.. The default transitions defined a collection of trees, with their transtions directed towards the root of the tree. This makes it possible to identify sets of transitions that generate the largest space reduction. But, choosing the default transition that produces the largest possible reduction where no cycle is created by the default transitions remained a question.

The solution relied on modeling the problem as a maximum weight spanning tree problem in an undirected graph, also referred to as a space reduction graph. Again, based on the criteria that every vertex has only one outgoing default transition, it implied that an arbitrary vertex could be picked to represent the root of the default transition tree with all the default transitions directed towards the root state. The problem with the said procedure was that it generated too many long paths on the DFA when implemented on a typical network application. This implies that the final D^2FA will require many transitions for every symbol it consumes. As a result with the selected tree root state centrally located, only some improvement was made, while the problem of having many long default paths remained prevalent. Becchi and Crowley (2007a, p. 147) showed that the problem of long default paths could be mitigated if none of the default transitions in a D^2FA lead from a state with depth d_i to another state with depth d_j, with $d_j \geq d_i$. This ensured that any string of length N will only need at most 2N state traversals to

be processed, thereby guaranteeing a 2N time bound on all D^2FA having only "backwards" transitions, where N is the length of any given string.

However, by constructing a maximum weight spanning tree with specified bounded diameter of 1, it still remained unclear if that will lead to producing the smallest D^2FA. The construction of such a tree is hard in a nondeterministic polynomial time (Hopcroft, Motwani and Ullman 2001, pp. 419-420). Kumar et al. (2006) realised that what they required was a collection of bounded trees of maximum weight. But, while the problem still requires a polynomial time if the diameter bounds is assigned the value 1, which is typical of any maximum weight matching problem, the problem still remained NP-hard for larger diameters.

Kumar et al. (2006) further employed the use of Kruskal's algorithm (Kruskal 1956, p. 48; Hopcroft, Motwani and Ullman 2001, pp. 414-419). Kruskal's algorithm constructs a minimal weight spanning tree from a given connected graph, with distinct positive real numbers attached to each edge of the graph. In the algorithm, edges are examined in decreasing order of their weights. The method also ensures that the addition of any selected edge does not create a tree whose diameter exceeds a specified bound. Afterwards the tree edges are used to define default transitions. In order to minimise the distance to the selected root from any leaf of any tree, the default transitions are made to point towards the roots of each spanning tree.

To properly estimate the reduction objectives of the D^2FA, a term called duplicate transition was introduced. Transitions were considered to be duplicates, when more than one of such transitions leads to the same next state for the same input symbol (Kumar et al. 2006, p. 345). After constructing the minimal state DFA from the selected regexps, both the normal and refined versions of the spanning tree are then used to construct the D^2FA. Table 3.1 shows the default transition paths having a path length of 4. The table also shows that the refined spanning tree yields relatively less complex D^2FA in comparison to the normal spanning tree. Column 1 of Table 3.1 represents the various content inspection DFA engines that have been constructed from the group of regular expression sets belonging to each separate engine. The Cisco regexp group containing 590 regexps originally had 97873 normal spanning trees on its DFA. After applying the refined version of the spanning tree with a bounded default paths of 4 edges, there was a 27.67% reduction in the total number of spanning trees in the D^2FA produced. Bro648 regexp group recorded the highest reduction after refinement with about 32.15% reduction. The Snort11 group of regexps recorded the lowest refinement with just 1.21% reduction in the total number of spanning trees.

Table 3.1: Number of transitions in D^2FA with default path length bounded to 4 (Kumar et al. 2006).

DFA	Normal spanning tree	Refined spanning tree	% Reduction
Cisco590	**97873**	**70793**	**27.67**
Cisco103	**115654**	**82879**	**28.34**
Cisco7	**37520**	**36091**	**3.81**
Linux56	**69437**	**66739**	**3.89**
Linux10	**314915**	**302112**	**4.07**
Snort11	**180545**	**178354**	**1.21**
Bro648	**11906**	**8078**	**32.15**

In summary the D^2FA reduces the memory requirements at the expense of multiple memory accesses. Also splitting a regexp set into constituent groups only adds to the number of memory accesses and generates more D^2FAs, all of which need to be processed in parallel (Kumar et al. 2006, p. 346). However, by using a memory-based architecture to implement the D^2FA, the multiple-embedded memories were used to map each D^2FA in such a way that a character is processed in one memory cycle only. Embedded memories provide ample bandwidth, which further reduces the space requirement as observed by Kumar et al (2006). This is made possible by splitting the regexps into multiple groups and creating a D^2FA for each of them. The shortcoming of the D^2FA is that it introduces an additional cost of several memory accesses for each input character. This is because the D^2FAs may need multiple default transitions to consume a single character. However, to ensure that the memories do not become a throughput bottleneck, each D^2FA was mapped to each memory in a way that there was minimal fragmentation of the memory space. This kept each memory uniformly filled, with each receiving a fairly equal number of accesses. Furthermore, the embedded memories give some added flexibility in the face of frequent changes made to the regexps during each update.

c. Delta Finite Automata

This section reviews the orthogonal DFA approach called the Delta Finite Automata (δFA) described by Ficara, Giordano and Procissi (2008). The δ in δFA is an indication of the differences between adjacent states. The δFA machine is capable of reducing the number of states and transitions in a traditional DFA, by requiring only one transition per character. This further reduces the overall memory requirements, while enhancing the speed of the δFA machine. The novel state encoding scheme of the δFA machine was tested in packet classification.

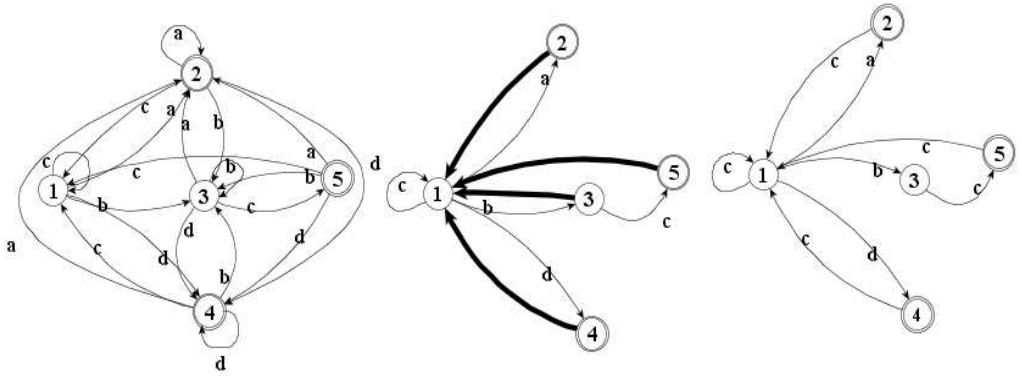
Ficara, Giordano and Procissi (2008) began by observing that most adjacent states on many DFAs share many common transitions. The shareable feature makes it possible to eliminate most of the redundant transitions, while leaving behind only the unique ones. Most orthogonal DFA methods trade-off size for fewer number of memory accesses per input character. However, DFAs could still be made a lot faster by taking advantage of smaller fast memories such as caches, while becoming smaller in overall size. This is possible with the right scheme in place as observed by Ficara, Giordano and Procissi (2008)
The D^2FA automata implemented by Kumar et al. (2006) (refer to Section 3.2.1-ii) inspired the creation of the δFA scheme. Figure 3.8a, and 3.8b are the same figures originally presented by Kumar et al. (2006) as shown in Figure 3.7. The same figures are used in this section for the purpose of comparison. You may recall that a default transition was introduced between states with similar outgoing transitions in the D^2FA shown in Figure 3.7.

A higher memory compression was achieved by applying some refined diameter-bounded maximum weight spanning tree algorithm. The same default transitions are similarly represented by thick lines as shown in Figure 3.8b. Furthermore, the δFA as shown in Figure 3.8c retains the advantages of the D^2FA, requiring only a single memory access per input character. Figure 3.8a represents a DFA on the alphabet {a,b,c,d}, which recognise the regexps (a^+), (b^+c) and (c^*d^+) (Ficara, Giordano and Procissi (2008, p. 33). Figure 3.8b also shows a D^2FA for the same regexps set as that of the DFA in Figure 3.8a.

The δFA scheme was designed to reduce the memory footprint of the states. This is achieved by keeping in memory a limited number of transitions for each state. Ficara, Giordano and Procissi (2008) observed that adjacent states shared most of the next states associated with the same input character. For

instance, if we jump from state 1 to state 2 as shown in Figure 3.8a, and we are able to remember the whole transition set of state 1 using a local memory, then all the transitions defined in state 2 are already known as well. The knowledge gained concerning the transitions defined in each state eliminates the need for loops on same character transitions. Also, because each character leads to the same set of states as state 1, it means state 2 can then be described with fewer amount of bits. However a jump from state 1 to state 3, where the next character to be consumed is a c, will create a different set of transitions, starting from state 1 as shown in Figure 3.8a.

The result of what is described as shown in Figure 3.8c is the δFA equivalence of the DFA in Figure 3.8a, excluding the accompanying local transition set. The δFA now has 8 edges as opposed to the 20 in the DFA of Figure 3.8a and the 9 edges in the D^2FA of Figure 3.8b. All the input characters of δFA require only a single state traversal as opposed to that of the D^2FA



(a) The DFA (b) The D^2FA (c) The δFA

Figure 3.8: Automata recognising (a+), (b+c) and (c*d+) (Ficara, Giordano and Procissi 2008).

Algorithm 3.1 describes the pseudo-code for creating δFA from an N-state DFA, given a character set of C elements. The algorithm works with the transition table represented by t[s,c] of the input DFA. The t[s,c] is an NxC matrix, having a row per state (2008). The i^{th} item in a given row stores the state number to reach upon reading an input char i. The end result is the compressible state transition table represented as t_c[s,c] used to store for each state, the required transition by the δFA alone. Any other cell of the t_c[s,c] matrix is filled with the special LOCAL_TX symbol which is taken care of using a bitmap data structure such as the one by Becchi and Cadambi (2007, p. 1065). A bitmap is considered to be a data structure used to represent the merged states and their respective transition labels. Refer to Section 3.2.1d for more details on a bitmap.

Constructing the δFA machine requires a step for every transition (C) of each pair of adjacent states (NxC) in the input DFA (2008). The δFA requires a time complexity cost of $O(NxC^2)$ time and O(NxC) space, because the structure is based upon an NxC matrix. The t_c matrix is first initialised with EMPTY symbols by the algorithm, which then copies the first root state of the original DFA into the t_c. The t_c acts as a base for the continuous storing of the difference between the consecutive states. Furthermore, because some states have identical sets of transitions, only the transition set of one of the states will need to be stored and the rest deleted. All the references to those single states left are then substituted. The process is repeated until the number of duplicate states becomes 0. The number of transitions per state is reduced due to the efficiency of the algorithm.

```
for c ← 1, C do
      t_c[1,c] ← t[1,c]
end for
for s ← 2, N do
      for c ← 1, C do
            t_c[s,c] ← EMPTY
      end for
end for
for S_parent ← 1, N do
      for c ← 1, C do
            S_child [s,c] ← t[S_parent, c]
            for   y ← 1, C do
            if t[S_parent, y] ≠ t[S_child, y] then
                  t_c [S_child, y] ← t[S_child, y]
            else
            if  t_c [S_child, y] == EMPTY
                  t_c [S_child, y] ← LOCAL_TX
            end if
              end if
             end for
         end for
 end for.
```

Algorithm 3.1:Pseudo-code for the creation of the transition table tc of a δFA from the transition table t of a DFA (Ficara, Giordano and Procissi 2008).

The lookup operation in a δFA as expressed in Algorithm 3.2 starts by reading the current state together with its entire transition sets. The corresponding entries to each transitions defined in the set read from the state is then updated in the local storage (Ficara, Giordano and Procissi 2008). Lastly, the next state s_{next} is then computed simply by observing the proper entry in the local storage t_{loc} as follows:

```
procedure Lookup (s, c)
read (s) /* which is implied*/
for i ← 1, c do
      if t_c[s,i] ≠ LOCAL_TX then
      t_loc[i] ← t_c[s,i]
     end if
end for
S_next ← t_loc[c]
return S_next
```

Algorithm 3.2: Pseudo-code for the lookup in a δFA. The current state is s and the input character is c (Ficara, Giordano and Procissi 2008).

Figure 3.9 shows the internals of the lookup example for the δFA as shown in Figure 3.8c. The circle marked state 1 of Figure 3.9 represent state 1, while its internals include a bitmap to specify which transition set are identified. The bitmap and transition set were defined during construction as explained by Ficara, Giordano and Procissi (2008). Given the input string "abc", the machine starts with t = 0 in state 1, and then copies the completely specified transition set into the local transition set. Figure 3.9

shows that input character a is first read and then the machine moves from state 1 to state 2 (t=0 to t=1). The move to state 2 specifies a single transition in the direction of state 1 on input of character c. Moreover, state 2 is an accepting state (underlined in Figure 3.9). When moving to state 3 on input character b, the transition taken is not specified within state 2, but is kept in the local transition set. State 3 now has a single transition specified also, but this time the one in the local transition set changes to 5. When input character c is read, the machine moves to state 5, which again is accepted and specifies a single transition toward state 1. The result is as shown in Figure 3.9 thus:

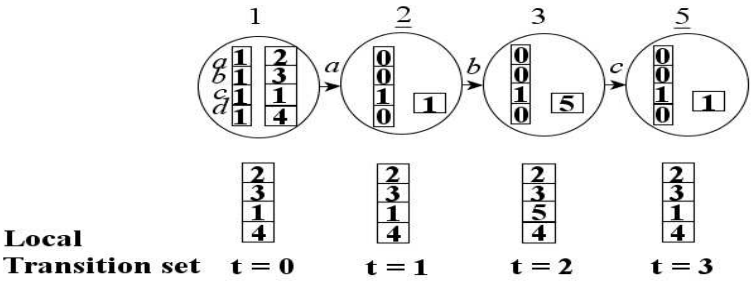


Figure 3.9: δFA internals: a lookup example (Ficara, Giordano and Procissi 2008).

In summary, the design described by Ficara, Giordano and Procissi (2008) compresses well for a composite DFA having no complex regexp patterns consisting of wild-cards. Also, the ability to copy transition sets enables quick retrieval and reuse of state information across the states, leading to further reduction in memory requirement.

d. Non-equivalent States Merging with Bitmap Compression

The approach by Becchi and Cadambi (2007, pp. 1064-1066) involves the merging of several non-equivalent states within a DFA. The scheme simply assigns labels on the input and output transitions of the DFA. The concept of equivalence of states was initially described by Lin, Tai and Chang (2007) in more detail in Section 3.2.1b-i. A bitmap based data structure that represents the merged states and their respective transition labels was also introduced in the scheme by Becchi and Cadambi (2007). Current rules contain more complex regexps, and when implemented as composite DFAs they place a high demand on memory. The complexity of the patterns further makes software regexp search engines slow and non-scalable. The argument then remains whether or not a reduction in the memory requirement for DFAs could increase their speed of matching. There is also the question of whether the DFAs could become more scalable and easier to implement within software regexp engines or specialised hardware architectures.

The approach by Becchi and Cadambi (2007, p. 1066) places no requirement on the transition between two states reaching a common destination as opposed to the DFA methods described by Kumar et al. (2006). The scheme implemented by Kumar et al. (2006) does not perform state merging, but rather attempts to eliminate redundant transitions to common destinations. The process of eliminating the redundant transitions is based on the criteria that two states reach the same destination, and have the same transitions based on a given input character. A description of a compact DFA data structure representing a DFA with merged states and transitions was described based on Algorithm 3.3 and Figure 3.10. The use

of a bitmap data structure for string matching on DFAs, which uses pointer indirection, was also introduced.

The data structure in Algorithm 3.3 shows that each state is made up of a structure having next state pointers for each of the 256 ASCII characters. A bitmap-based data structure was used to maintain pointers to valid next states in the transition table. The data structure uses a bitmap indexed by the input character to generate an address into the transition table. The bitmap eliminates the need to maintain explicit next state pointers as descried by Becchi and Cadambi (2007). A value of '1' in the i^{th} position of the bitmap index indicates a valid next state transition by the input character having the value i. Also, by counting the number of 1's until the i^{th} bit is reached in the bitmap, the address into the related transition table is obtained. A '0' on the other hand in the bitmap shows that the next state pointer does not exist in the transition table. This makes the transition default to the start state. Algorithm 3.3 shows the algorithm that uses a naïve data structure to represent a state in a DFA.

```
The DFA_state {
    RegExList*accepted_regexp:
    unsigned int*bitmap[8]:
    DFA_state*next_state[256]:
    DFA_state*failure;
  }
```

Algorithm 3.3: Basic naïve data structure representing a state in a DFA (Becchi and Cadambi 2007).

Figure 3.10a shows the bitmap for state 3 as shown in Figure 3.11. The lower part shows the bitmap and its transition table for state 3 from the example in Figure 3.11. The figure shows that the number of entries in the transition table is equal to the number valid outgoing pointers from state 3 which is 5. Also, the number of five 1's in the bitmap tallies with the effective outgoing transition from state 3 (which are characters 'a', 'f', 'g', 'h' and 'i'). However, it shows that the basic bitmap-based data structure illustrated in Figure 3.10a does not eliminate duplicate entries in the respective transition table. Becchi and Cadambi (2007) designed a more straight forward approach for eliminating the problem of duplicate entries.

The solution required the use of a second bit for all locations in the bitmap as shown in Figure 3.10b. The bit is used to indicate whether or not the address into the transition table must be incremented. However, there was a high memory requirement for alphabets that required large bitmaps due to their large cardinality. To mitigate the memory problem, one level of pointer indirection table was inserted. The table was placed between the bitmap and the transition table containing only three distinct next state entries. The bitmap then generates an address into the pointer indirection table, which in turn holds a pointer into the transition table. The pointer indirection table only requires 2-bits, and has a width that is O(log n), where n is the number of distinct next states. As a result sixteen such entries will require to be packed into a 32-bit memory. However, the two shortcomings of a bitmap-based data structure (Becchi and Cadambi 2007) are as follows:

i. Counting of 1's is necessary for processing the bitmap and for obtaining an address into the transition table, but it was time consuming.
ii. Loading up the bitmap needs numerous memory accesses.

It was acknowledged that the two issues could be resolved to a certain level by splitting a large bitmap into several smaller bitmaps, with each appended with some additional comment. Further observation also shows that a state in a DFA built from most pattern rules rarely have valid outgoing transitions based on all possible characters in the alphabet set. Rather, the state has only a few valid character class transitions to valid next states.

Another observation was that the DFA has most of its transitions directed towards a default (failure) state which is the initial state. Notwithstanding, non-equivalent states could be merged and relabelled to produce a compact DFA with a reduced memory requirement and increased scalability. The labels 0 and 1 as seen within black the squares appearing on transitions going into states 3 and 4, represent all input transitions of state 3 and 4 as shown in Figure 3.11. All transitions that are not shown, lead to state 0 in the figure. Characters in the range such as [g-i] labelled as transitions into state 5 indicate that any of the characters 'g', 'h', or 'i' could make a basic transition to the destination state 5, with state 6 as the accepting state. Considering Figure 3.11 as a motivating DFA example, a non-equivalent state merging technique was introduced to further reduce the size of the DFA in Figure 3.11. The process involved the merging of two or more non-equivalent states.

```
The DFA_state{
RegExList*accepted_regex;
unsigned int*bitmap[8];
DFA_state*transition_table[num_valid_transitions
DFA_state*failure;
}
```

```
Struct DFA_state{
RegExList*accepted_regex;
unsigned int*bitmap[8];
unsigned int*ptr_indirection[[num_valid_transitions
DFA_state*transition_table[num_valid_transitions];
DFA_state*failure;
}
```

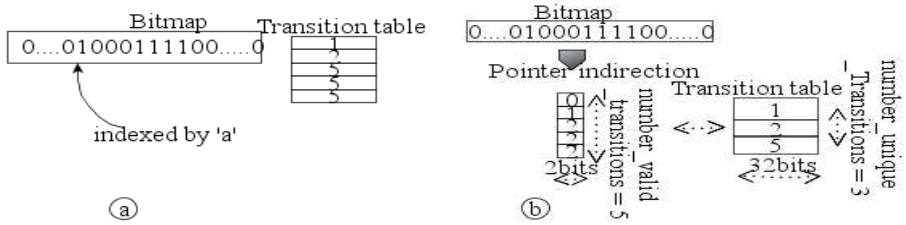


Figure 3.10: (a) Rudimentary bitmap-based data structure, (b) More compact bitmap-based data structure using pointer indirection (Becchi and Cadambi 2007).

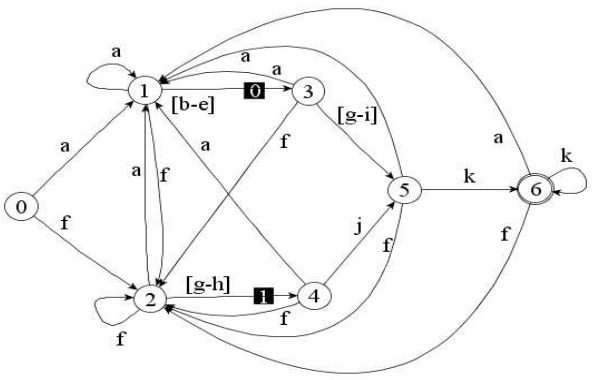


Figure 3.11: DFA for regexp (a[b-e][g-i]|f[g-h]j)k+. (Becchi and Cadambi 2007).

Figure 3.12 shows how the non-equivalent states 3 and 4 were merged together. The merging of the two states was possible even though the affected states do not transit to state 5 on the same input character. However, that is not a requirement for state merging in such a scheme. Also, each transition arc is denoted by the character on which the transition occurs followed by the transition label as shown in Figure 3.12.

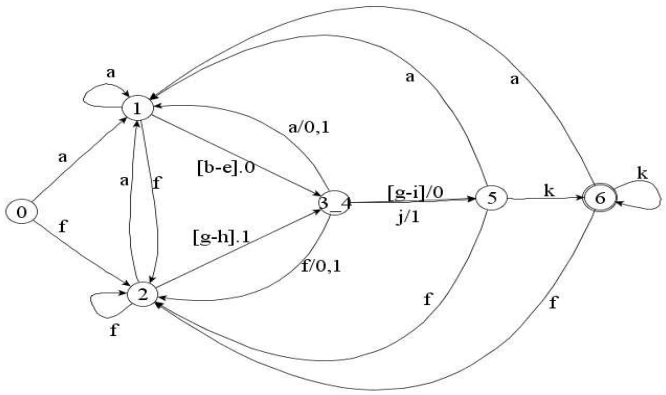


Figure 3.12: DFA after merging states 3 and 4 (Becchi and Cadambi 2007).

The first task to be performed while merging two states is to label their transitions, which is necessary as merged states share the same data structure. During DFA traversal, when such a data structure is accessed, it is necessary to monitor how a destination state was reached. It is also necessary to be aware of which portion of the data structure to access. The only restriction in labelling is that all input transitions of a state from the original DFA must have the same label as described by Becchi and Cadambi (2007).

After merging states 3 and 4, the new merged state is relabelled as state 3_4, with its output transitions represented with trailing labels. The transitions [g-i]/0 and j/1 indicate that next state 5 is the same state reached from state 3_4, on inputs g, h or i, with label 0, as well as input j with label 1. Transitions a/0,1 and f/0,1 show that a transition is made upon consuming characters ‘a’ and ‘f’ irrespective of the label used to reach state 3_4. As shown in Figure 3.13, transitions of the type a.0/0,1 now appear, where the label after the dot (.) relate to the destination state and the label after the ‘/’ symbol relate to the source state of the transition. The transition a.0/0,1 from state 3_4 back to state 1_2 demonstrates that:

i. The transition carries with it a label 0 that specifies its destination state 1_2, which is meant for the underlying original DFA state 1.
ii. The transition is taken when its source state 3_4 receives a label 0 or 1.

With non-equivalent states 1 and 2 also being merged, the DFA shows how the outcome looks like as shown in Figure 3.13.

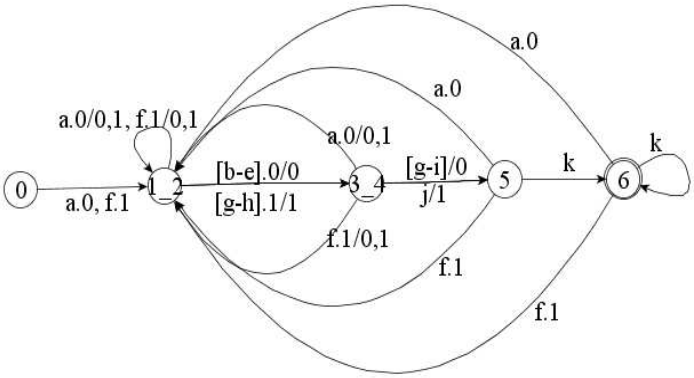


Figure 3.13: DFA after merging states 1 and 2 from the example of Figure 3.12 (Becchi and Cadambi 2007).

The data structure for the merged state 1_2 as shown in Figure 3.14 uses the same bitmaps depicted in Figure 3.10 prior to the merging of the two states as shown in Figure 3.14. The pointer indirection table indexes the transition table, where the next state pointers are stored. The table then supplies the destination labels for the outgoing transitions (Becchi and Cadambi 2007, p. 1068). The data structure does not require storing the source labels because they are implicit. However, state merging requires that the data structures of all the affected states should be updated.

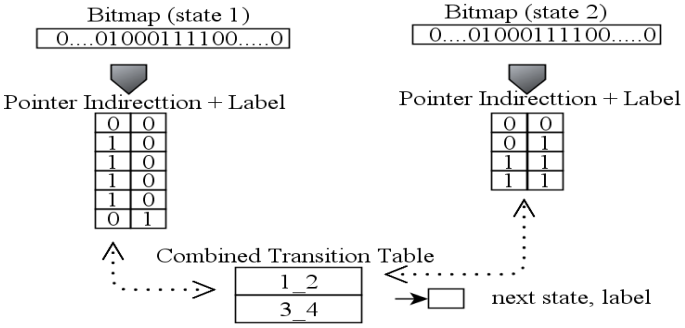


Figure 3.14: Merged data structure for the state 1_2 of Figure 3.13 (Becchi and Cadambi 2007).

Generally, two or more states can be merged to form a single state by introducing labels on their transitions. The states that are merged can then be represented using a data structure that holds the separate bitmaps. The structure is a combined transition table made up of the union of their separate transition tables. The structure also consists of the updated pointer indirection table for every one of the original DFA states, as well as a structure to store the labels. A description of the algorithms for creating the destination labels, merging and labelling was also explained by Becchi and Cadambi (2007, pp. 1068-1070), details of which are not relevant to this thesis.

In summary, the approach describes a scheme that substantially reduces the memory footprint of DFAs while retaining the speed with guaranteed worst-case performance. This was achieved by using the concept of the non-equivalence state merging of a DFA. This is based on a transition labelling technique, with label reuse. Another important advantage of the scheme is that by merging states together, more common destinations for other states are formed, thereby opening up more chances for further merging of states, which translates to more memory reduction. However like most DFA approaches affected by

complex regexp patterns consisting of wildcards and constrained repetitions, the approach here has not discussed the effect of such patterns on the design.

Table 3.2 gives a summary of the approaches described in Section 3.2. The summaries highlight the pros and cons of each of the approaches discussed in Section 3.2.1.

Table 3.2: Summary of all approaches discussed in Section 3.2.

Approach	**Summary**
1. String matching approaches:	
a. Compile-Search Algorithm by Thompson (1968).	Pros: The approach utilised a compile search algorithm to create a context-search compiler. The compiler then generates an object language from an acceptable regexp considered as the source language. A signal is generated whenever an embedded string in the given text matches a given regexp. Cons: The approach is not suitable for complex regexp pattern matching.
b. A Sub-Linear Search Algorithm by Boyer and Moore (1977).	Pros: The algorithm searches for the characters of a given pattern in a text string precisely once. With the information gained by starting a search at the end of the pattern, the algorithm is able to make significant leaps through the given text that is searched. The search is made in worst case time O(nm) and average case time O(n/m), where m is the pattern length and n is the text length. Cons: The approach is only suitable for string pattern matching.

Table 3.2: (*cont'*d).

Approach	Summary
2. Regexp matching approaches	
a. Pseudo-Equivalent State Merging Technique by Lin, Tai and Chang (2007).	Pros: The approach describes a technique that first compiles the matching string patterns into a FSM. The output of the FSM is asserted whenever any substring of the input strings matches the string pattern. The approach is efficient in reducing the number of states and transitions belonging to an original FSM, which ends up reducing the overall required memory size. Cons: Cycle problems created during the merging of FSM states limits the amount of pseudo-equivalent states to be merged. As such, the fewer the number of patterns that can be merged, the more the limitation placed on the memory saving objective.
b. Delayed input FA (D^2FA) by Kumar et al. (2006).	Pros: The approach is able to reduce the overall memory requirement by incrementally replacing several transitions from a given DFA with a single default transition. Cons: The D^2FA introduces an additional cost of several memory accesses for each input character. This is because the D^2FAs may need multiple default transitions to consume a single character.
c. Delta Finite Automata (δFA) by Ficara, Giordano and Procissi (2008).	Pros: This is a special finite automaton (FA) that exploits the differences between two adjacent states. The idea is based on the fact that most adjacent states on DFAs share common transitions. The ability to copy transition sets enables the quick retrieval and reuse of state information across the states. This leads to further reduction in memory requirement. Cons: The use of a bitmap as the data structure of choice becomes a bottleneck as the number labelled transitions belonging to merged states grows.
d. Non-Equivalent State Merging with Bitmap Compression by Becchi and Cadambi (2007).	Pros: The approach is built upon a bitmap based data structure which is indexed by the input character to generate an address into the state transition table. The approach places no restrictions on the transitions between two states reaching the same destination. Furthermore, by merging non-equivalent states, more common destinations for other states are formed. Cons: Processing the bitmap in order to obtain an address into the transition table is time consuming. Also, loading up the bitmap needs numerous memory accesses.

3.2.2 FPGA-Based Approaches

An FPGA device is a highly reconfigurable device. Also, the fine-grained parallelism provided by an FPGA device can reduce the space and processing time requirement (Sidhu and Prasanna 2001) of most NFA designs, if well exploited. The approaches for high throughput regexps pattern matching exist for evaluation based on DFAs or NFAs (Yu et al. 2006; Tripp 2008; Sidhu and Prasanna 2001; Mitra, Najjar and Bhuyan 2007; Wang et al. 2010, p. 209). A hybrid-FA approach (Becchi and Crowley 2007b) also exists. The limited parallelism provided by serial processors such as multi-core CPUs (Yu et al. 2006) makes it suitable to implement DFA designs, but inefficient for NFA designs. NFAs are better suited for devices such as FPGAs, because of the parallelism provided by such a reconfigurable device.

There are quite a number of FPGA-based approaches which were implemented in different ways (Brodie, Taylor and Cytron 2006; Mitra, Najjar and Bhuyan 2007; Jiang and Prasanna 2009; Becchi and Crowley 2008; Tripp 2006; Chaves et al. 2008; Wang et al. 2010; Singapura et al. 2015). One of the earliest known form of NFA implementation that uses logic was the one implemented by Floyd and Ullman (1982, pp. 603-604). In the logic approach, a regexp was converted into an NFA by first parsing the regexps. Afterwards, the McNaughton-Yamada algorithm (McNaughton and Yamada 1960, p. 41) was employed to recursively produce NFAs for regexps: $R_1 + R_2$, R_1R_2, and R_1*, where R_1 and R_2 are any arbitrary regexps. However, the reconfigurable design constructed in this thesis also exploits the logic provided by an FPGA device. A description of the design is first introduced in Section 3.2.2f, but fully described and implemented in Chapter 5.

a. Pattern Matching Designs using Self-Reconfiguration

i. Fast Regular Expression Matching using FPGAs

Sidhu and Prasanna (2001) describe an algorithm that automatically parses a regexp into its constituent sub-expressions. The algorithm then applies the ϵ-NFA rules described by Floyd and Ullman (1982, pp. 607-608) to construct an NFA that matches the same strings as the given regexp. The states of the NFA described by Sidhu and Prasanna (2001) were constructed using a technique called one-hot encoding (OHE). The OHE technique uses a Flip-Flop (FF) to represent each state in the NFA. A bit value of 1 stored in each of the state FFs signifies that the state is currently active. The technique was also adopted by Baker and Prasanna (2004, p. 3).

Figure 3.15a – 3.15d shows the FPGA implementations for the ϵ-NFAs representing the regexps r_1 and r_2 described in Figure 2.1 – 2.3 of Section 2.3 respectively. Algorithm 6 implements the logic structures in Figure 3.15a - 3.15d. The i port of the top level logic structure is permanently high (bit value 1), since the given NFA is required to match strings beginning at any position in an input text. Another reason is because NFAs normally process a single string and determines if a match has occurred or not. The o port is connected to a FF that represents the accepting state of the NFA.

Figure 3.15a, describes the logic structure for the NFA that matches a single character, with the FF relating to the accepting state removed. Figure 3.15b describes the logic that implements the NFA for the regexps $r_1|r_2$. An OR-gate is used as the only logic to combine the outputs of the NFAs N_1 and N_2 representing the regexps $r_1|r_2$. Figure 3.15c describes the logic that implements the NFA for the regexp r_1r_2, which requires only three wired edges. Figure 3.15d shows the logic implementation for the regexp r_1^*. An OR-gate was used as the logic that combines the output of the two inputs in order to generate the output for the accepting state.

It was further observed that while implementing NFAs as logic, if all the source input FFs to the destination input FFs is on ϵ–transitions, then the FFs can be removed without being implemented at all. Also, because ϵ–transitions from accepting states only help in building bigger NFAs, then the FFs relating to the accepting states can also be removed. This is illustrated as shown in the logic structures of Figure 3.15a - 3.15d for the given regexps r_1 and r_2. The arrows in the figures are wires.

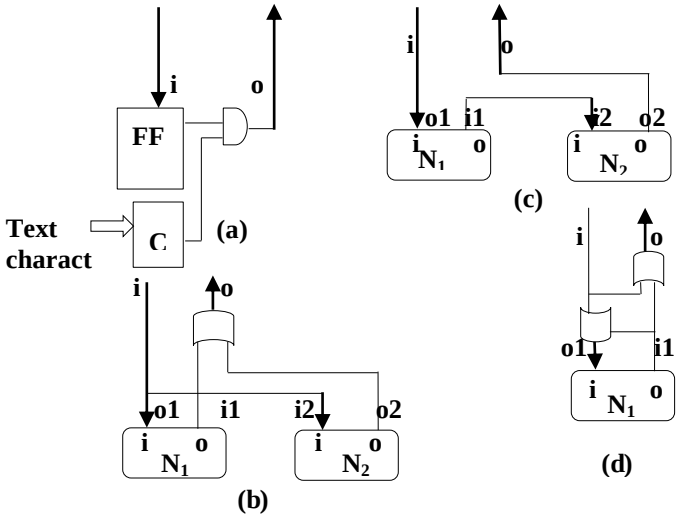


Figure 3.15: Logic structures for the regexps (a) single character, (b) $r_1|r_2$, (c) r_1r_2, (d) r_1*. (Sidhu and Prasanna 2001).

During the NFA construction, the algorithm accepts the regexp in postfix form. The postfix form removes the need for the "AND" metacharacters and helps to streamline the algorithm. The postfix form is obtained by performing a postorder traversal of the syntax tree of a given regexp. The algorithm makes use of a stack data structure and depends on the following placement and routing subroutines: place_char, place_|, place_., place_* as seen in Algorithm 3.4. The subroutines place the logic structures accordingly for a character. The metacharacters: char, |, . ,*, return a pointer reference to the placed structure (Sidhu and Prasanna 2001, pp. 230-231). The placement and routing subroutines take a constant O(1) time, while the algorithm is constructed in O(n) time, where n is the length of the regexp. The constructed NFA also processed one character each clock cycle in O(1) time. Although it took only O(1) time to process a character on a serial machine, it required about $O(2^n)$ time to construct the equivalent DFA.

Implementing the NFA construction algorithm on FPGA architecture using self-reconfiguration requires that the NFA mapping time should be kept as small as possible. Self-reconfiguration refers to the ability of a device to generate configuration bits at runtime and also be able to modify its own configuration. This makes it possible for an NFA to be constructed as a configured logic on the device itself. The device that that can be configured to perform self-reconfiguration is referred to as a Self-Reconfiguration Gate Array (SRGA). The mapping time consists of the time required to construct the NFA and produce configuration bits for the NFA logic (Sidhu and Prasanna 2001).

Furthermore, it is essential that the overall mapping time should be minimal, because the NFA is constructed only at runtime during the user input of the regexp. The NFA construction algorithm was implemented as a program that took a regexp and generated its output as NFA logic. The output is in the form of an HDL description and technology mapped netlist (refer to Section 2.7). The mapped netlist is then placed and routed or straight away made to produce the required configuration bits (Xilinx 2010, pp. 136-137; Sidhu and Prasanna 2001, p. 232). Algorithm 3.4 generates the corresponding NFA from the

regexp ((a|b)*) (cd) using Self-Reconfiguration implementation. Algorithm 3.4 performs the placement and routing operations. It is vital that the logic cell in which a given logic structure is described should be properly configured. By design, the pre-incremented counters row and col, together with registers row1, col1, row2 and col2 are used for the purpose of placement and routing. The stack pushes or pops a 'row, col' pair in a single operation. Each of the switch cases for |, *, ., and char represent the subroutines. The clock cycles indicated on the right hand side of Algorithm 3.4 indicate the time it takes to configure the logic structures for each of the subroutines: char, |, . ,*,.

```
                                                          Clock Cycles
1  row=I; col =0; i=0;                                        [1]
2  while(i<regexp_len)
3  {
4   switch(regexp[i])
5     {
6       case char: place_char (regexp[i], col);              [46]
7           push (0, col);                                    [1]
8                   ++col;                                    [0]
9
10      case   | :  pop(&row1, &col1);                        [1]
11                  pop(&row2, &col2);                        [1]
12                  place_|(row,col2);                       [22]
13                  route_row(col2, col1);                   [10]
14                  route_col(row, row2);                     10]
15                  push(row, col2);                          [1]
16                  ++row;                                    [0]
17
18      case   . :  pop(&row1, &col1);                        [1]
19                  pop(&row2, &col2);                        [1]
20                  place_.(row,                             [14]
21                  route_row(col2, col1);                   [10]
22                  route_col(row, row2);                    [10]
23                  push(row, col2);                          [1]
24                  ++row;                                    [0]
25
26      case   * :  pop(&row2, &col2);                        [1]
27                  place_*(row, col2);                      [26]
28                  push(row,col2);                           [1]
29                  ++row                                     [0]
30
31     }
32   ++i
33  }
34       pop(&row, &col);                                     [1]
35       route_input_high(row, col);                          [3]
36       route_output_high_ff(row, col);                      [3]
```

Algorithm 3.4: NFA construction algorithm using Self-Reconfiguration (Sidhu and Prasanna 2001).

In summary, the approach by Sidhu and Prasanna (2001) introduced a self-reconfiguration process. The process improved the construction time for an NFA by several magnitudes. The design accelerated the NFA process of automatically generating the required netlist. The netlist is then implemented and used to produce the configuration bits ready for loading the design onto a target FPGA device. The approach also set the pace for several other approaches that consider building reconfigurable designs. However, the approach needed to be extended further to perform multi-character and multi-pattern matching (Sourdis and Pnevmatikatos 2004; Brodie, Taylor and Cytron 2006; Yamagaki, Sidhu and Kamiya 2008, Singapura et al. 2015).

b. Multi-Character Regexp Matching Designs

The concept of multi-character matching was explained by Becchi and Crowley (2008, p. 52) based on the scheme called k-DFA. The value k is the number of characters of a given input string consumed per single clock cycle. For a given DFA defined over an alphabet Σ, the non-compressed k-DFA has $|\Sigma|^k$ outgoing edges per state. The ability to make the k-DFA perform alphabet-reduction and implement default transitions (Becchi and Crowley 2007a, p. 147; Kumar et al. 2006) gave it a distinct advantage.

The concept of alphabet reduction is motivated by the example in the FSM scheme described by Brodie, Taylor and Cytron (2006, p. 194). The scheme uses only a small subset of the entire alphabet. The benefit of the default transition relates to its ability to remove all redundant transitions relating to DFAs This is achieved by taking advantage of the various state transitions driven by the same input character (Becchi and Crowley 2008; Kumar et al. 2006). This ensures that if the stride doubles, the number of states will be guaranteed to stay the same, even though the number of transitions increases very quickly.

Bispo et al. (2007, p. 178) reviewed the current status and open issues regarding the synthesis of regexps targeting FPGAs. It was observed in their review that, multi-character matching per clock cycle was a viable option for a pattern matching scheme to achieve high clock frequency. However, with overlapped matching involved, such a design can lead to undesired results as all possible byte alignments (Clark and Schimmel 2004) must be put into consideration. The scheme that addressed the problem of overlapped matching was implemented by Yu et al. (2006). Also re-write rules were used to create more efficient regexps by eliminating the problem of repeated searches, and ensured that fewer passes were involved in the matching processes.

A scalable pipeline architecture, which extended the AC machine to create an extended machine called Aho-Corasick DFA (AC-DFA) (Jiang, Yang and Prasanna 2010, pp. 3-6) was implemented. The AC-DFA converts a given string pattern with n characters into a DFA with n states. Thus, the AC-DFA takes O(m) time to process any given input stream that contains m characters. Beginning at the root of the AC-DFA, to the end of each pattern, a state is added per character to the tree, by each pattern. With the traditional AC-DFA, a single memory block holds only one active state at a time. The AC-DFA normally has cross transitions generated based on failure transitions, which differ from the traditional AC-DFA. The AC-DFA is made up of only a few staged processes aimed at finding the minimal pipeline depth of the DFA. The depth of a state on the AC-DFA is the directed distance from the root to that state. The minimal depth was achieved by removing most of the cross transitions in the AC-DFA rather than eliminating all the failure transitions like the other pipelined solutions do. Removing all the failure transitions can only lead to a deep and non-scalable pipeline design as observed by Jiang, Yang and Prasanna (2010).

Furthermore, in order to increase the throughput of the AC-DFA capable of performing multi-character matching, a compressed AC-DFA was built by Alicherry, Muthuprasanna and Kumar (2006, pp. 189-190). This version of the compressed AC-DFA can process W ($W \geq 1$) characters per clock cycle, where W is the input width of the character input. The scheme divided each pattern into W-byte blocks used to construct the AC-DFA (Jiang, Yang and Prasanna 2010, p. 7). Afterwards, the W instances of the compressed AC-DFA were executed in parallel.

i. Parallel String Matching Engine for a NIDS

Tripp (2006, p. 21) describes an approach where a FSM operates on a single byte wide data input. The approach assigns a different FSM for each byte wide data path from a multi-byte input data word. The given search strings are first split into many interleaved substrings. Subsequently the outputs from the separate FSMs are then combined in a way that enables string matching to be performed in parallel across multiple FSMs. The approach builds upon the approach designed by Tripp (2005, p. 28), where a standard table-based FSM implementation was preceded by a preliminary multi-byte compression process that uses classification of inputs. The compression process reduces input data into a series of tokens, designed to run at a rate of one word per clock cycle.

The approach described by Tripp (2005) explained the sharing of pre-FSM classifiers requiring some high logic cost for implementation. The approach provided a unique solution for resolving priority conflicts that may likely occur when some input data matches more than one of the patterns under consideration. Such an occurrence was attributed to the presence of wildcards in the patterns, which divided the ordering into two parallel systems. By using ‘primary’ classifiers, all strings (Tripp 2005, p. 31) are matched separately from those terminating with real characters. Those strings starting with wildcards are matched using the ‘secondary’ classifiers. The approach described by Tripp (2006) was also designed to match the start and ends of strings which have a probable chance of occurring part way through a streaming data word. The process also requires the matching of the parts of the start and end string having wildcard characters. Higher performance was achieved by constructing a FSM that could match multiple bytes of input characters per clock cycle.

Furthermore, given a w-byte wide input word, w different FSMs each of which is trying to match all w instances in the substrings consisting of a w-way interleaved search string was constructed as shown in Figure 3.16a. Each FSM has a w-bit match vector (Tripp 2006) output specifying the substrings matched per clock cycle. It follows that, if all the w substrings appear in a given order through the all w FSMs at the appropriate time, then a match is reported to be found for the search string. Figure 3.16b shows the set of search substrings with $w = 4$. With the occurrence of each of the last 4 bytes of the search string likely to relate to the instance when each of the related substring reports a match. Furthermore, a string alignment process was employed to tackle that.

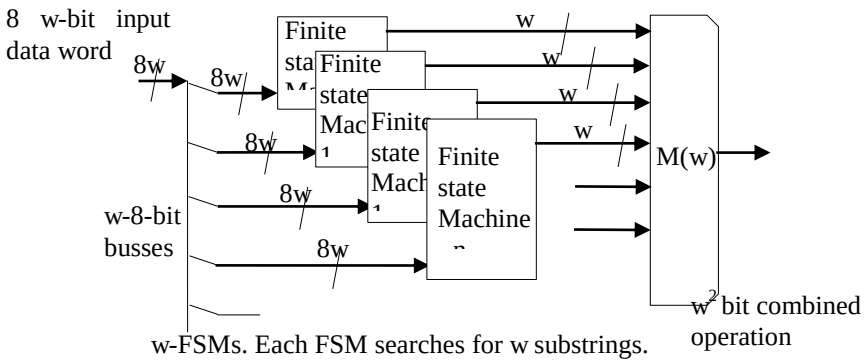


Figure 3.16: (a) Matching interleaved substrings (Tripp 2006).

Word size = 4

Search string =	"the -cat-sat-on-the-mat "	
Substring 0 =	" e t t - - "	= "ett --- "
Substring 1 =	" - - - t m "	= "----tm "
Substring 2 =	" t c s o h a "	= "tcsoha "
Substring 3 =	" h a a n e t "	= "haanet "

Figure 3.16: (b) Interleaved substrings of the search string "the-cat-sat-on-the-mat" (Tripp 2006).

However, the problem associated with table based compression is usually that of significant memory requirement. For instance, a Mealy machine with s states, i input bits and o output bits, the memory M requirement in bits could reach:

M= $(\lceil \log_2 s \rceil + o).2^{i+\lceil \log_2 s \rceil}$ (Tripp 2005, p. 28).

The rest of the approach describes how to minimise the memory size by creating a packed array (Tripp 2008, p. 4), which could be implemented in an FPGA. Each entry also contains the base address of the state vector in the packed array for deciding the next state. The algorithm amounts to significant memory savings for larger FSMs since it avoids the use of two dimensional arrays.

In summary, the approach could be further improved if the memory requirement for the state decoding can be further decreased by exploiting the redundancies that exist within the table. This could be achieved by exploiting the transition redundancies, using a form of memorisation based on a local transition set that can be attached to each state of the w FSMs.

ii. Increased Striding with Run-Length Coding Scheme

Brodie, Taylor and Cytron (2006) described a novel FPGA-based based pipelined FSM. The FSM employs encoding and compression techniques in order to improve the capacity and speed of character matching. The compiler decodes a set of regexps and places them in optimised design structures.

The compiler circuit is first given a set of regexps, and then the regexp circuit implements the high throughput FSMs. Alphabet encoding was used to minimise each of the FSMs resources. High throughput and encoding was combined together to produce the symbol Equivalence Class Identifiers (ECI) encoding blocks. This was achieved by employing directly addressed tables and pairwise combination. The FSM's tables were then compressed and the resulting transition information is deployed in the Indirection and Transition Tables (ITT) (Brodie, Taylor and Cytron (2006, pp. 192-193).

The size of the table posed a practical limitation to the length of an input sequence that can be contained in any computing platform with fast storage. The index into the encoded table is called an ECI. An algorithm of $O(|\Sigma|^k|Q|)$ divides Σ (alphabet) into a reduced set of ECIs (called ECDs in the design implemented in this thesis). The algorithm was used to form the columns of the given transition table δ: Q x $\sum \rightarrow$ Q, and the index into the encoded table is called an ECI. The FSM described by Brodie, Taylor and Cytron, constructs the pattern \$[0-9]+(\.[0-9]{0,1} with symbols: $, ., [0-9], and all other ([^$.0-9]) symbols not in the pattern (2006). The symbols are then represented by the corresponding ECIs: 0, 1, 2, and 3 respectively (Brodie, Taylor and Cytron 2006).

The logic for converting a sequence of m symbols into an ECI suitable for presentation to a transition table became necessary for processing an input stream at high throughput. Furthermore, a direct technique for performing such an operation would require executing pairwise combinations using directly addressed tables. Theoretically, it means the method could be used to allocate an ECI to an arbitrary number of input symbols. Also, memory efficiency significantly reduced as the number of symbols covered in the final ECIs increased. Furthermore, with increased stride came the problem of more memory requirement. The striding process involves the consumption of multiple symbols in a single clock cycle by the FSM. The algorithm responsible for increasing the striding process is invoked repeatedly. Each time it is invoked, it doubles the number of symbols processed per clock cycle. Furthermore, to remedy the memory problem attributed to the increasing memory requirements, Brodie, Taylor and Cytron employed the use of a run-length coding scheme. The coding scheme is used to cut down the storage needs for a sequence of symbols that display sufficient redundancy.

However, the coding scheme is only efficient whenever a significant amount of repetitions is found on a given input string. The idea behind the scheme was to be able to code a string say a^n, where n is the run-length and a is the symbol using the notation n(a). For instance, the string "aabbbbaaaaaaabbbaabbaaa" is coded as 2(a)4(b)7(a) 3(b)2(a)2(b)3(a). As such, if each symbol and run-length requires a byte of storage, then the example given would reduce the total amount of bytes needed from 23 bytes to 14 bytes only. The equivalent run-length coded transition table for the pattern \$[0-9]+(\.[0-9]{0,1}is as shown in Figure 3.17. The 3-tuple entry 5(B,1,0) as seen in column 0, for state A of Figure 3.17 is read thus: n(next state, restart flag, accept flag). The value n is the run-length value. The restart flag indicates a restart transition, while the accept flag indicates an accept transition. The next state is the state transited to upon consuming the ECI value of 0 (which is the $ character). For instance, the 3-tuple 5(B,1,0) indicates that there is a transition from state A to B on the input of the character $, which is to be restarted and not accepted. Even though the column compression technique saves memory, it also intensifies the cost of memory access to the transition table in order to obtain a chosen entry. For instance, from the ECI value 2 column of Figure 3.17, we can see that the run-length code for 1(D,0,1) is repeated twice. To solve the problem, a technique was introduced that uses indirection and transition tables (Brodie, Taylor and Cytron 2006).

States	ECI			
	0	1	2	3
A	5(B,1,0)	3(A,1,0)	1(A,1,0)	5(A,1,0)
B			1(D,0,1)	
C			1(E,0,0)	
D		1(C,0,0)	1(D,0,1)	
E		1(A,1,0)	1(A,0,1)	

Figure 3.17: Run-length coded transition table (Brodie, Taylor and Cytron 2006).

The technique for the run-length coded table was optimised to use pre-computed prefix sums. The scheme places compressed columns in physical memory efficiently. The indirection table has a pointer used for reading the first memory word from the Transition Table Memory (TTM). An index was also created, which serves as the address where an entire column is accessed, by reading w successive words from the TTM. The w successive words are computed with x being the number of entries per memory word (2 * 5 entries per word, for a dual-port memory) as follows:

$$w \cong \left\lceil \frac{count+index}{x} \right\rceil$$

The index and count values determined which entries in the first and last memory words were used in the column. The indirection table and the TTM were organised in a given way, and used by the state select block of the design (Brodie, Taylor and Cytron 2006, p.197). Further optimisations were later made to the final indirection table and the TTM. The optimisation helped to minimise the number of memory accesses to the run-length coded TTM using a unique memory packing scheme.

In summary, the approach is suitable for implementation in FPGAs. However, the FPGA implementation produced a throughput of 4Gbps, and clocked only at 133MHz. The overall design was also limited by accommodating no more than a 1000 REMEs.

c. Common Prefix, Infix and Suffix Matching Designs

The approach described by Hieu et al. (2011, p. 108) compiles regexps with a view to optimising the hardware resources available in a target FPGA. Firstly, a technique which involved the sharing of common infixes in regexps was implemented. The design has the ability to recognise overlapped matches (Yu et al. 2006, p. 96). Secondly, the technique that constructed five adaptive building blocks was also described. The block designs combined the separate building blocks together to create the required hardware for a given set of regexps. Thirdly, a technique which automatically generated regexp pattern matching engines straight from the Snort rules database was also developed.

The regexp matching architecture as shown in Figure 3.18 is composed of four main modules. An incoming input character obtained from the data FIFO is supplied to the character matching module in every clock cycle. This ensures that all current characters and character classes are matched. The output signals of the character matching module are then supplied to all the regexp matching engines (REMEs) residing within the PCRE matching module. Every REME is made up of a regexp constructed from some building block (BB). The common prefix matching process is implemented inside the PCRE matching module, while the common infix sharing process is handled separately in the infix matching module. Lastly, all matching signals are collated and encoded through the encoder module (Hieu et al. 2011).

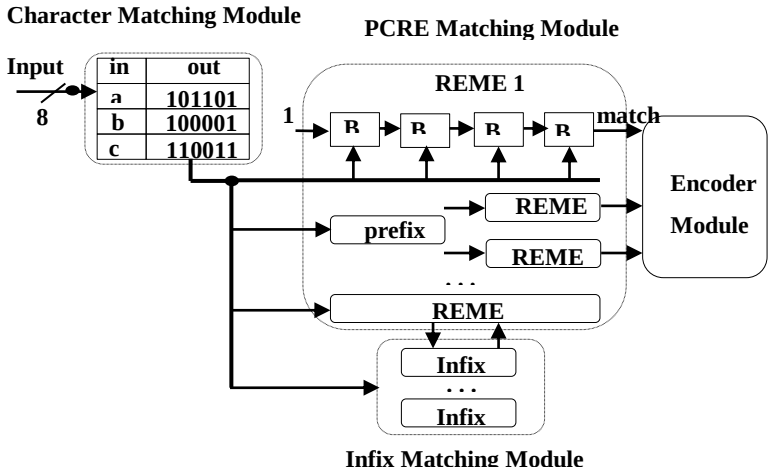


Figure 3.18:Top-level diagram of regexp matching system (Hieu et al. 2011).

Extracting common prefixes shared between multiple regular expressions was achieved by building a modular NFA in a step-wise order (Hutchings, Franklin and Carver 2002; Sourdis and Pnevmatikatos 2004; Becchi and Cadambi 2007; Yu et al. 2006; Lee et al. 2007; Lin et al. 2006). In the NFA, all regexps are converted and joined together with the same start state (Hieu et al. 2011). The next step involved applying the algorithm to reduce all modular NFA states which share common accepting character and incoming states.

However, with the infix sharing process there was a difference, because care is needed in the design to avoid false positive matches. This is because the infix is situated right in the central position of the pattern, which necessitated keeping track of the previous sub-match information. In order to illustrate the concept, two regexps "abcdef" and "cdcdeg" were considered the example of patterns which share a common infix sub-pattern "cde" (2011). The implementation directly routes the output of the infix block to the two remaining sub-patterns. This guaranteed that all matches would be met, with the possibility that it could also lead to a false positive match. For instance, considering an input string like "abcdeg", the match result will be reported against the pattern "cdcdeg", which should not be. The reason is that, the previous successive match prefix was lost at the infix sharing block. As a result, by the time the output of infix block is made high (signal = '1'), it will be difficult to establish which of the two patterns the signal originated from (Hieu et al. 2011; Lin et al. 2006).

In order to address the problem with the infix issue, Hieu et al. (2011) designed a new sharing scheme. The new scheme kept track of all previous matches and extensively handled the problem of overlapping matches as shown in Figure 3.19. The scheme operated by replacing the common infix by a shift module in each regexp. The shift module was constructed using a k-bit shift register (Xilinx 2012c), with k as the length of the infix. The output of both the infix and shift module blocks are then ANDed before sending it to the suffix block, in order to match a signal from the prefix sub-pattern.

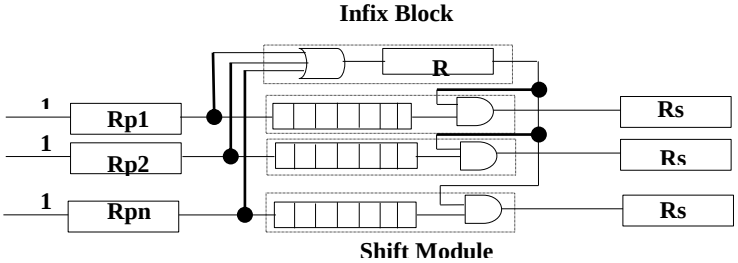


Figure 3.19: The new infix sharing architecture (Hieu et al. 2011).

The logic cost attributed to the use of 8-to-256 decoders (Clark and Schimmel 2003, p. 957; Clark and Schimmel 2004, p. 251) for handling character classes can be high (refer to Section d). However, the output signals from each of the decoders indicate that one of the 8-bit characters is matched. The increased cost of logic attributed to the decoder is due to the high number OR gates required when character classes are involved. Hieu et al. (2011) thereby utilised a Block RAM Centralised Character Matching (BCCM) scheme to handle character class matching instead. The BCCM has a depth of 256, with a width that is reliant on the number of unique characters in the regexp set. Data is then read as a bit vector of 256-bits, with each bit in the vector considered to be a match for the individual NFA states.

Accordingly, separate bits are automatically transmitted to applicable BB matching circuits. However, in the scheme it meant that each character matching circuit costs one 256-bit column in BRAM. While it is efficient for character class inputs, it will be a waste of memory for single character inputs. However, the memory waste could be mitigated by allowing only a single instance of similar characters in memory and sharing (Hieu et al. 2011) matching signals among the various REMEs. This will guarantee that the scheme will have no redundancies attributed to combinational logic. This will also reduce the number of LUTs utilised for building the block circuits.

Further optimisations were also performed to gain more efficiency in the PCRE to NFA hardware implementation scheme such as the re-writing of regexps with constrained repetitions (Long et al. 2011, Yu et al. 2006). The process involved re-writing patterns with a small number of repetitions such as: $n<3$, where n is the number of repetitions. Without such optimisation effectively in place, the process could lead to a lot of logic and memory requirements. The optimisation is especially required when utilising a counter-based scheme like the one described by Long et al. (2011, pp. 67-68). For instance, an expression like a{3}, could be easily replaced by the simple expression “aaa” instead (Hieu et al. 2011, p.109). Also a single character option in an expression like “(a|b|c)” could easily be replaced by the character class “[abc]”, thereby eliminating the need for use of three OR-gates.

In summary, the scheme sacrificed throughput for a reduction in the logic circuit cost. This is in comparison to the other decoding approaches. This implied that fewer LUTs and FFs costs are incurred, while enjoying the benefit of an average throughput. Moreover, there is always a trade-off between having to increase throughput and obtaining lower logic circuit costs with such approaches.

d. Shared Character Decoding Designs

i. Extended Shared Decoding Scheme

The scheme described by Clark and Schimmel (2004, p. 249) is a more scalable FPGA approach used for pattern matching. The approach implemented by Clark and Schimmel is capable of sustaining a throughput in the ranges of 1 - 100 Gbps. The approach offered a trade-off between density and

throughput (2004). The density refers to the number of utilised logic circuits used within a limited circuit area. There is a significant advantage attributed to pattern matching in hardware such as FPGAs. However, performance degrades rapidly if such patterns are implemented in software-based designs (Becchi and Crowley 2007b; Yu et al. 2006; Kumar et al. 2006; Lin, Tai and Chang 2007).

The scheme described by Clark and Schimmel (2004) retained and built upon the character shared decoding approach described by Clark and Schimmel (2003). The approach by Clark and Schimmel (2003) substituted comparators for decoders during implementation. However, the approach described by Clark and Schimmel (2004) achieved a higher throughput, by processing multiple input characters in parallel (Brodie, Taylor and Cytron 2006; Sourdis and Pnevmatikatos 2004; Tripp 2006; Sourdis and Pnevmatikatos 2003) as shown in Figure 3.20. To process n characters per clock cycle, the pattern matching circuit required n decoders. Each of the decoders decodes separate input characters per clock cycle. A higher throughput is guaranteed by the approach, but density was sacrificed in exchange for a higher throughput.

By shifting n character inputs at once, a pattern may start at any given location within each of the n-character matching blocks. The approach made it necessary to scan for all patterns at every n possible offsets. Furthermore, n parallel NFAs were required by every pattern matcher to find pattern matches at all offsets. Each pattern matcher has an output that indicates when a match is obtained at any of the offsets. The pattern matches are aggregated through an AND gate. After the last packet is processed, k-match signals are then collected in a k-match vector. The signals represent the outputs from the various NFAs within the various pattern matching blocks. The match output signals are then forwarded for onward packing into a 32-bit word output encoder (Clark and Schimmel 2004, p. 252).

Every logic element (LE) in the design could implement at most a four-input logic gate and a FF. As such, a single LE was capable of matching up to four characters at once. The formula for computing the upper bound on the number of FPGA logic elements was proposed. The formula depended on the length of the pattern l, and the input width n. With the size of n greater than l, extra logic savings was achieved (Clark and Schimmel 2004, p. 252). This is because, the match function does not have more than $l+1$ inputs. More savings are achieved if the size of n increases, since $l+1$ remains the same. The ratio of LEs per pattern matcher, when n is greater than l is summarised as:

$$\text{LEs/Matcher} = \left\lceil \frac{l+1}{4} \right\rceil \times \left(\left\lceil \frac{l-1}{n} \right\rceil + 1 \right) \times n$$

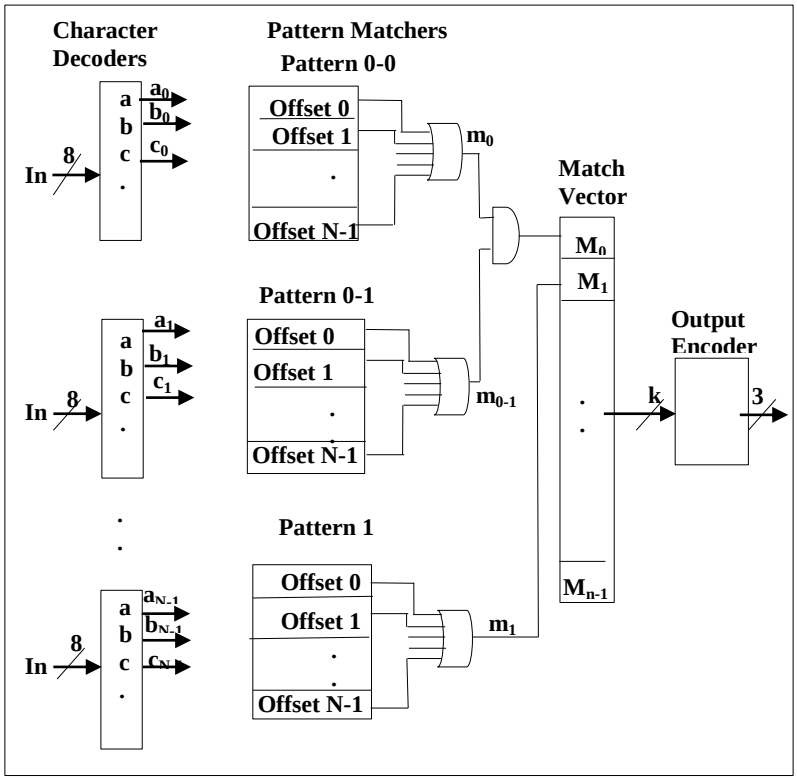


Figure 3.20: Pattern matching module using a multi-character decoder NFA (Clark and Schimmel 2004).

In summary, the approach is quite beneficial in terms acquiring higher throughput. This was achieved by using a shared character decoding approach to perform multiple character matching. However, by increasing the number of patterns matched, the design will be impractical for matching patterns having wildcards and character classes with constrained repetitions (Becchi and Crowley 2007b; Yu et al. 2006) and (Long et al. 2011). This is because, such patterns are capable of increasing the memory bandwidth and other logic circuits required, and that will be a major bottleneck to such a design.

e. Regexp Matching Engine Designs

i. Systematic Translation of Compact Matching Engines

The software by Yang and Prasanna (2009) systematises the translation of regexps into compact and high-performance regexp-NFAs (RE-NFAs) (Yang, Jiang and Prasanna 2008). Given a fixed number of fan-out transitions per state an n-state, m-byte-per-cycle REME (Mitra, Najjar and Bhuyan 2007; Yang, Jiang and Prasanna 2008; Ganegedara, Yang and Prasanna 2010) is built in O(n x m) time and requiring O(n x m) memory. The whole circuit realised occupied no more than O(n x m) slices (Xilinx 2012a, p. 7) on the FPGA. However, Yang and Prasanna (2012) used a clever algorithm to create a spatially stacked n-state RE-NFA with max fan-out d in O(n x d) time, and produced a circuit that only utilised O(n x m x d^2) area. Thus recursively, Yang and Prasanna (2012) was able to construct an n-state m-character REME in O(n x d x $\log_2$m), starting from a one-character matching circuit for an n-state RE-NFA.

The focus of the design developed by Yang and Prasanna (2009) centred on the automatic parsing, conversion and construction of REMEs implemented by Yang, Jiang and Prasanna (2008). The REMEs were constructed using the regular expression matching, nondeterministic finite automata (RE-NFA)

architecture for a completely automatic FPGA implementation. The process involved in the construction of the REMEs was based upon the following components as outlined by Yang and Prasanna (2009):

i. Dynamic translation from regexp parse tree to an even and segmental RE-NFA.
ii. Programmed construction of RTL (refer to Section 2.7) codes in VHDL. This involved the creation of a spatially stacked circuit, a configurable number of times for multiple character matching (Brodie, Taylor and Cytron 2006; Sidhu and Kamiya 2008).
iii. Distribution of centralised character classification (Hieu et al. 2011; Long et al. 2011, p. 72) in BRAM, which is designed for up to 256 REMEs based on trivial heuristics (Yang and Prasanna 2009).
iv. Programmed construction of up to 16 pipelines in a two-dimensional layout.
v. Development of a 'benchmark generator of regexps with configurable pattern complexity parameters [namely] state count, state fan-in, loop back and feed-forward distances' (Yang and Prasanna 2009).

The REME construction is in three phases: (1) parsing of the regexps into tree structures, (2) use of the modified McNaughton-Yamada (MMY) construction for creating the RE-NFAs, and (3) mapping of the RE-NFAs into structural VHDL suitable for FPGA implementation. The MMY parsing process required no ϵ-transition-only nodes to be introduced, as described by Yang, Jiang and Prasanna (2008, pp. 32-33). The process of converting the regexp parse trees to NFAs retained the MMY used by Yang, Jiang and Prasanna (2008, p. 32). During the process, a selection rule was applied thus:

i. Only patterns of average lengths were selected, and patterns that are too short or simple were avoided such as: `ab{2}` etc.
ii. Identical regexps in separate rules are considered to be one.
iii. Regexps with long repetitions were avoided, such as regexps containing "[^\n]{256}". As is the case with the own approach.
iv. Regexps requiring back references were also avoided.

The process of translating RE-NFA to VHDL (Yang and Prasanna 2009) required that each pair of nodes inside a lightly shaded ellipse is mapped to an entity statebit with one parameter as seen in Figure 3.21. The parameter is the number of input ports determined by the number of prior states that directly move to the present state. All inputs combine to a single OR gate within the entity statebit. This is trailed by character matching through AND logic, together with a state value register. The single bit output value attributed to the register is joined to the inputs of the immediate next states (Yang and Prasanna 2009) as shown in Figure 3.21.

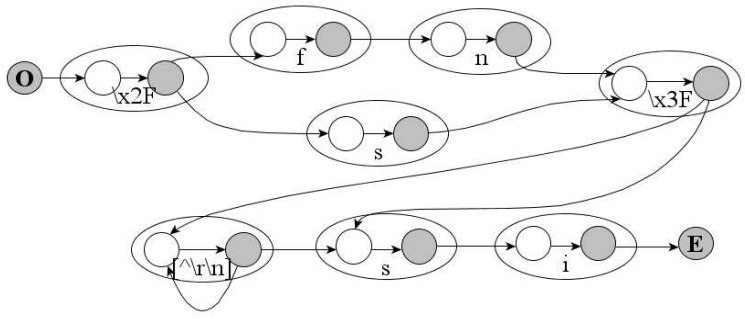


Figure 3.21: A modular NFA for "\x2F(fn|s)\x3F[^\r\n]*si" constructed using the MMY rules (Yang and Prasanna 2009).

The use of a BRAM-based character classifier to match any given 8-bit character class was described in the architecture by Yang, Jiang and Prasanna (2008) and Hieu et al. (2011). The classifier utilises a 256-bit column of BRAM for every input character class. A function is called to observe and to associate each state's character class to the character class entries stored in the BRAM. Every 1-bit outcome is then routed from the BRAM to the correct entity statebit as input to the AND gate. This makes it possible to implement an n-state RE-NFA where n > 1, on a single BRAM having no more than 256 x n-bits using a two-phase approach thus:

i. The first phase gathers a set of selected character classes from a regexp. A floating-point sorting key is then assigned each set. The assignment is based on the number of times a character class appears just once in the regex, and then its assigned sorting key becomes its index position within the regexp. Otherwise, the average of all its index positions within the regexp becomes the sorting key if the character class appears multiple times within the regexp.
ii. The second phase sorts the unique character classes based on their sorting keys and instantiates them as BRAM columns. Each column is assigned and linked with the identifier of the instantiated character class. Lastly, the output of each column in the BRAM is then joined to character inputs with the same identifier.

To effectively implement an automatic architectural optimisation, the REMEs are spatially stacked to form Multiple Character Matching (MCMs) circuits. Afterwards, they are grouped into clusters of 16. The clusters are then passed to a two dimensional staged pipeline structure (Yang, Jiang and Prasanna 2008, p. 35). The benefits of the spatial stacking approach include: simplicity with a moderate time complexity, and flexibility which is the ability to generate an MCM REME of any natural number. This is unlike the temporal extension approach described by Yamagaki, Sidhu and Kamiya (2008, p. 134), which only generated RE-NFAs with $m=2^i$, where m is the number of MCM REMEs, and i is the number of algorithmic iterations.

The process of carrying out REME clustering for staged pipelining involved improving and providing a solution to the shortcomings of the approach described by Yang, Jiang and Prasanna (2008). In the approach, a two-dimensional staged pipeline was implemented using a priority encoder at every stage to generate the pattern matching outcomes. However, the problem with such approach was that the process involved in arranging REMEs into a staged pipeline structure was difficult and error-prone when done manually. Another difficulty with the approach was attributed to the complex operations involved in the buffering and circulation of the MCM signals, represented by the thick arrows as shown in Figure 3.22. The figure is a 2-D staged pipeline design, with a total p pipelines and r stages per pipeline. Furthermore, there were problems of the routing complexity and varying resource utilisations arising as well. The problems led to performance variation between the REME clusters which is caused by the application of different REME grouping schemes.

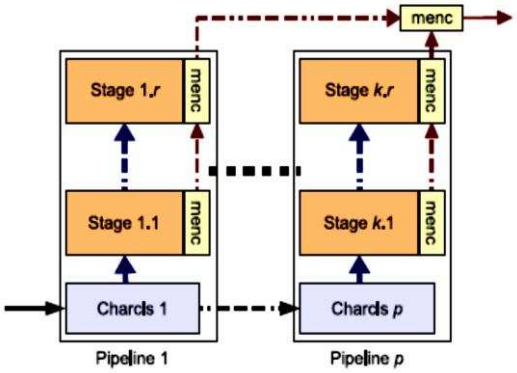


Figure 3.22: Structure of a 2-D staged pipeline (Yang and Prasanna 2009).

However, to overcome the problems mentioned, a simple heuristic was implemented to marshal k-REMEs with a total n-states and p-pipelines. The implementation was such that when adding a new REME to an existing pipeline, a function was called to relate each of the character classes in the current REME to the former accessed in the BRAM. Suitable links are then joined from the BRAM output to the inputs of the corresponding states whenever identical character classes are found. The time complexity of the procedure was estimated as O(k x n x w), where w is the number of unique character classes amid the n states in the k-REMEs, and the space complexity was estimated as O(256 x w).

It was observed that in actual applications, w grows almost linearly with respect to a trivial n, but rapidly levels and propagates much slower than O(log n) as n becomes reasonably larger by a few hundred. Lastly, the process of matching outputs from all REMEs was prioritised by assigning higher priority to lower-indexed pipelines and stages.

In summary, the staged and pipelined approach described by Yang, Jiang and Prasanna (2008, p. 35), upon which that of Yang and Prasanna (2009) was built is effective in scaling up the number of REMEs in a single circuit. The process kept a linear growth in LUT usage, which is in relation to the number of REMEs. However, the scheme left some room for further optimisations in order to accommodate more regexps having a higher number of character classes and pattern length. The growth of such regexps has an impact on the clock rate of the hardware design, due to the higher cost of state fan-in of the REME. Also regexps with long constrained repetitions usually take a toll on the amount of the LUTs and other related resources utilised by such designs considered in the approach.

f. Classification-Based Designs

The concept of classification was utilised by Brodie, Taylor and Cytron (2006), Gupta and McKeown (1999) and Tripp (2006) in their respective approaches. Arnold (2007, pp. 1-5) describes relations on a set of real numbers to include for instance: "=", "<", and "≤" etc. The symbol "≃", is used to denote an abstract or specific relation such as: a ≃b to mean '*a'* and '*b'* are related. This concept was further elaborated by Arnold (2007, pp. 1-5) with the following definitions:

i. A relation on a set S is a subset of S x S.
ii. Given that ≃ is defined over a given set S, the relation ≃ could be reflexive, symmetric, and transitive.
iii. A relation ≃ on the same set S is said to be an equivalence relation if all three conditions in (ii) hold.
iv. Given that ≃ is an equivalence relation on a set S. Then ∀ a ∈ S, an equivalence class of a, is represented by $[a]$ as the set: $[a] = \{x \in S \mid x \simeq a\}$. (Arnold 2007).

Ilie, Solis-Oba and Yu (2005, p. 312) implemented an NFA reduction scheme that used the concept of equivalences (Lin, Tai and Chang 2007; Becchi and Cadambi 2007). The scheme optimally reduced the size and search time of any given arbitrary NFA, and by using equivalence, NFA states were merged successfully. The concept of equivalence was built upon the initial concept of NFA reduction based on left and right equivalences described by Ilie and Yu (2002, p. 338). The approach made it possible for NFAs to be reduced in their right ($\equiv_R$) and left ($\equiv_L$) invariant equivalence. The right invariance is the largest invariant equivalence to the right of the NFA, while the left invariance is the largest invariant equivalence to the left of the NFA (Ilie and Yu 2002).

For instance, given any states say p and q, the distinguishability of the two states to the right, given any words that lead from p or q to the final states, was considered. The process can be performed symmetrically also to the left given any words that lead from the initial states to p or q. This is a form of 'reversed automaton, [where] say [in the] NFA M^R, all the transitions are reversed and the position of the initial and final states are interchanged' (Ilie and Yu 2002). Figure 3.23b shows states 1, 2 and 3 merged together because they belong to the same equivalence class of $\equiv_R$. Similarly states 4, 5, and 6 were merged because they belong to the same equivalence class of $\equiv_L$. However, Ilie, Solis-Oba and Yu (2005) observed that there is no exclusive way to use the $\equiv_R$ and $\equiv_L$ optimally. Figure 3.23d shows that the NFA from left has only two pairs of equivalent states: $1 \equiv_R 3$ and $1 \equiv_L 2$. It means that either states 1 and 2 or states 1 and 3 can be merged together, but not both. This is because merging states 1, 2 and 3 would introduce the word "bd" which does not belong to the language. An illustration on how equivalent classes are reduced using $\equiv_R$ and $\equiv_L$ is demonstrated in Figure 3.23a–3.23c and Figure 3.24.

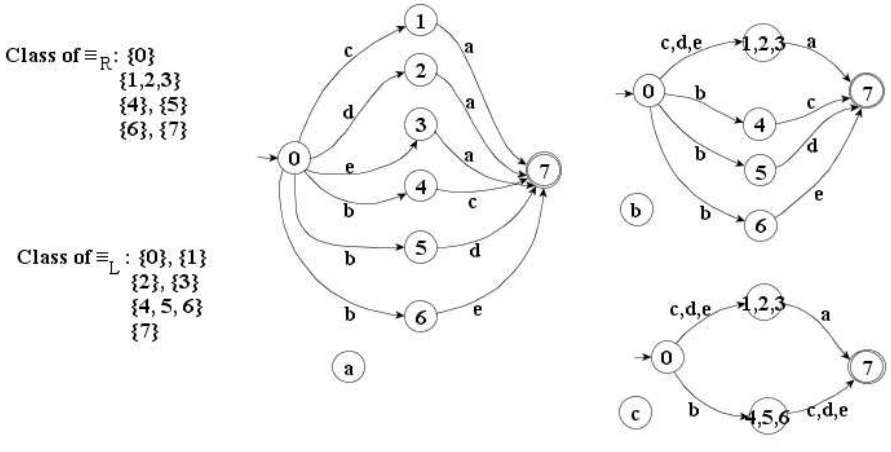


Figure 3.23: (a) An NFA, (b) The NFAs reduced version using$\equiv_R$, (c) The NFAs reduced version using both $\equiv_R$ and $\equiv_L$ (Ilie, Solis-Oba and Yu 2005).

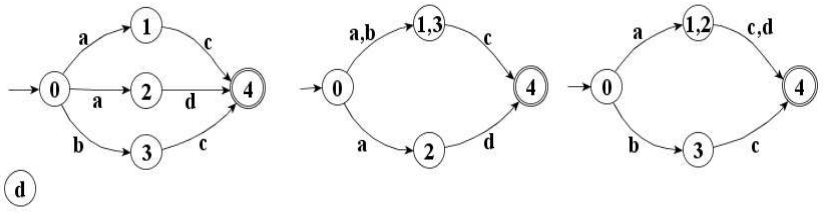


Figure 3.24: An NFA, and its reduced versions using $\equiv_R$ and $\equiv_L$ (Ilie, Solis-Oba and Yu 2005).

An NFA M = (Q, A, δ, I, F) describes an algorithm $O(|\Sigma|^k|Q|)$ that divides Σ into a reduced set of equivalence classes. This was achieved by forming the columns of the given transition table δ: Q x Σ → Q

to represent the classes (Brodie, Cytron and Taylor 2006). The index into the encoded table is called an ECI (refer to Section 3.2.2b-ii). Given a pattern \$[0-9]+(\.[0-9]{0,1}, the classes of characters: '$', '.', '[0-9]', and all other characters ([^$.0-9]) are created and represented by the ECI values 0, 1, 2, and 3 respectively.

Deep packet inspection (Becchi and Crowley 2008; Kumar et al. 2006; Kumar, Turner and Williams 2006) has become a critical function in various NIDS. The inspection process ensures that almost all the matching rules for any given packet are reported. An approach that was capable of resolving problems of priority conflicts was implemented by Tripp (2005, p. 31) (refer to Section 3.2.2-i). Priority conflicts are likely to occur due to the presence of certain input data. Such inputs are capable of matching more than one of the patterns under consideration, due to the presence of wildcards in the patterns. The solution involves dividing the ordering into two parallel systems. Afterwards, a 'primary' classifier is used to find all strings separately from those that terminate with real characters. A 'secondary' classifier is also used to find strings starting with wildcards (Tripp 2005).

The use of a BRAM-based character classifier to match any given character class of 8-bits by utilising a 256-bit column of BRAM was described in the architecture by Yang, Jiang and Prasanna (2008, p. 35), Hieu et al. (2011) and Yang and Prasanna (2009). Also, a novel state encoding scheme was implemented by Ficara, Giordano and Procissi (2008). The scheme was tested for use in packet classification, with focus on reducing the memory footprint required by DFAs (refer to Section 3.2.1c).

An approach based on a multi-stage classification was implemented by Gupta and McKeown (1999, p. 150). It was observed that quite a number of network services needed packet classification. Such services include: routing and access-control in network firewalls. A packet classifier is nothing but a set of rules which determines the class that a packet belongs to. A rule 'classifies which flow a given packet belongs to, based on the contents of the packet header(s)' as described by Gupta and McKeown (1999). The classifier is founded upon certain criterion on a given number of F fields of the packet header. The classifier associates to every class an identifier or classID. The classID is responsible for uniquely assigning the action associated with any given rule. Structure is expected in the classifiers which if properly exploited, can affect the classification algorithm used.

The classification process is necessary in order to decide which flow each arriving packet belonged to. This was necessary in order to properly forward and take the right action on such a packet. A good example of such an action is the filtering of the class of services to be apportioned to a given packet. The scheme described by Gupta and McKeown (1999) used a Recursive Flow Classification (RFC) technique. The technique exploits the structure and redundancies found in various network classifiers when matching rules on a network. The RFC process involves scanning of every single byte of packet header, with the aim of identifying predefined set of matching patterns. The classification algorithm is based on a simple heuristic algorithm called the RFC algorithm that has p phases as shown in Figure 3.25. Each p phase is made up of a set of parallel memory LUTs. Each of the table lookup is a reduction such that the value returned is shorter (in small bits) than the index of the memory access.

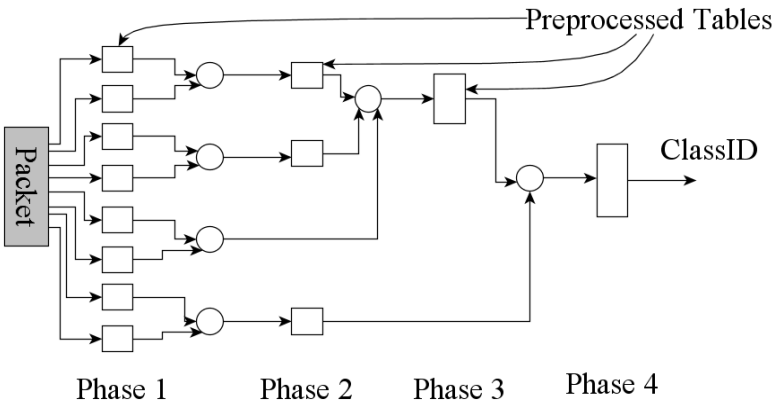


Figure 3.25: The packet flow RFC (Gupta and McKeown 1999).

While it was possible for applications having tables that frequently change to be incrementally updated, it was observed that the subject required further investigation. Each added phase of the RFC increases the amount of compaction on the original classifier. As such, the idea of adjacency grouping of two or more rules was then implemented in order to alleviate the problem.

Moreover, two rules are said to be adjacent if they are adjacent in some dimension. If indeed they are adjacent, then the two rules are merged to form a new rule. This is done in order to retain the specification of either of the two rules and the corresponding action associated with each rule. This is only achieved without further exploiting the redundancy and structure of the RFC. The setback with the RFC process is that it consumes too much memory, especially for classifiers with four fields in them.

In order to implement the classification-based approach implemented in this thesis, a type of state transition table that determines the set of next states for each active current state on the initial ECD-NFA was explained. Members of an equivalence class shall be identified as the ones having identical columns in the table. However, on this occasion the table columns cannot be held as columns of next states numbers anymore, but as vectors of next state numbers. Afterwards, a number of entries in the table contained empty sets.

i. The ECD-NFA Two-Phased Design

The ECD-NFA design implemented in this thesis originally describes a composite NFA based on the concept of equivalence classification as described by Arnold (2007), Brodie, Taylor and Cytron (2006), Tripp (2006), Gupta and McKeown (1999), and Ilie, Solis-Oba and Yu (2005). The ECD-NFA is constructed in $O(n^2m)$ time, requiring no more than O(nm) memory, where n is the pattern length of a single regexp, and m is the number of patterns compiled together (Yu et al. 2006). However, the design generates $O(\log_2 m)$ REMEs originally containing ECD-NFAs. The design also requires O(k) memory locations to store the generated k ECDs in the hardware BRAM. The REMEs are then arranged in parallel as shown in Figure 3.26, and not in the two-dimensional staged and pipelined approach described by Yang and Prasanna (2009). The block diagram shows the overall structure of the two-phased scheme before it was optimised.

The layout of the block design structure of the ECD-NFA is as shown in Figure 3.26. In the first phase of the processing chain, the program starts by compiling the regexp files extracted from the Snort rules. The regexps are then processed by the parser module. The module then calls the ECD-NFA

function generator to recursively create the n-byte ECD-NFA, where n = 2 and 4. The ECDs and their respective transition tables, as well as the BRAM modules are then generated. The ECD transition tables and the BRAM tables are then populated with the required ECDs.

Furthermore, the ECDs generated are guaranteed to grow steadily at O(k), where k is the number of ECDs created. The ECDs are created by performing the cross product computation of two 1-byte ECD inputs of vectors of next states (or simply state vectors). The cross computation generates a new 2-byte state vector containing the union of the vectors of next state. The vectors are generated based on each given ECD input (refer to Section 5.4.1a for details) on the automata. The process is iterated until n = 4 before it is halted, and the results of the 4-byte ECD-NFA are then translated into a synthesis-ready VHDL file. The required VHDL files are then interfaced with the second phase of the processing chain as shown in Figure 3.26. The full details of the operations, methods and processes involved in executing each of the two phases in the structure is discussed fully in Chapter 5.3.The summary of the block design in Figure 3.26 (refer to Section 5.4.2a for more details) is as follows:

a. During the process of creating the ECD-NFA units in the second phase of Figure 3.26, each of the ECDs generated represents a class of inputs that trigger transitions representing unique state vectors. The state vectors represent sets of vectors of next states, transited to from any current state(s). Each of the ECDs represents those sets of inputs that have the same effect on the automata.
b. The design then recursively performs a cross product computation of two ECDs state vectors, to generate an n-byte compressed ECD input transition table, using a similar tree-like RFC table structure described by Gupta and McKeown (1999), with n = 2 and 4.
c. Once the n-byte table of ECDs are generated at each stage, the table of n-byte ECDs is then synthesised into logic. The synthesis process involves the use of a simple but fast algorithm in the hardware design phase contained in the second phase of Figure 3.26. An optimised algorithm is later implemented which eliminates the use of decoders, registers and other associated logic circuits completely (refer to Section 5.4.2a-ii for details). This gave rise to the optimised version of the ECD-NFA design called ECD_RTS-NFA.
d. The VHDL files representing the design are then passed as the output of the compiler process in the first phase of the design tool chain. With the VHDL files generated, a 28-bit input is supplied to the n-REMEs arranged in parallel in the second phase of the design toolchain as. At the end of the second phase of the implementation, a 4-bit vector of 1-bit matches corresponding to each parallel REME pipeline is generated for encoding.

The need to reduce the construction and synthesis time of the ECD-NFA, led to the creation of ECD_RTS-NFA. The ECD_RTS-NFA being the optimised version retains the basic structures of the ECD-NFA, but has increased performance and reduced logic circuit requirements (refer to Section 5.4.1 for more details).

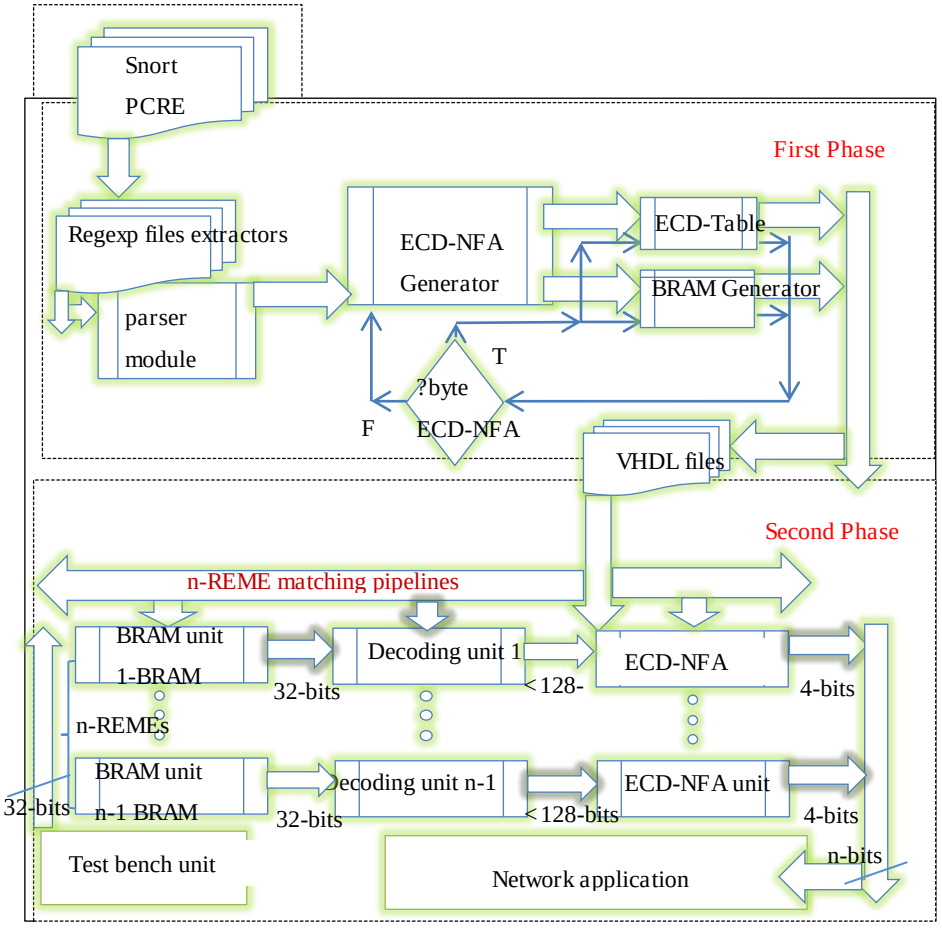


Figure 3.26: The ECD-NFA two-phased toolchain design Block diagram.

The major motivations and details of the approach are also discussed in detail in Section 5.1 and 5.3. The biggest problem that has been associated with most classification based approaches cost of losing the originating source of the original classifier. An example is the RFC memory saving scheme described by Gupta and McKeown (1999). This is because while the classifier can correctly decide the action for every newly arriving packet, it cannot correctly decide which rule in the original classifier it matched. This problem is attributed to the fact that the rules have been recursively merged over time while creating adjacency groups. This implies that the distinction among the rules is lost and thereby no longer distinguishable. The lack of distinguishability makes such approaches inefficient for implementing pattern-specific matching systems. However, the classification approach in thesis does not lose any information while creating the relevant 4-byte BRAM and ECD transition tables.

The initial ECD-NFA design incurs a significant timing cost, due to the initial use of 7-bit input decoders with 128-bit wide registers. The table look up operation in the decoding modules performs a nested loop operation which is not suitable for the VHDL synthesis tools to execute. In fact, it makes it almost impossible to synthesise the design at all in the second phase of the design toolchain. The process timing issue inspired the need to eliminate the logic overheads attributed to the decoding units, by constructing the optimised version of the design namely ECD_RTS-NFA (refer to Section 5.4.2a-ii for more details).

Table 3.4 gives the summary of the approaches described in Section 3.2. The table also highlights the pros and cons of each of the separate FPGA-based approaches discussed in Section 3.2.2.

Table 3.4: Summary of FPGA-based approaches discussed in Section 3.2.2.

Approach	Summary
1. Fast Regular Expression Matching using FPGAs by Sidhu and Prasanna (2001).	Pros: The approach parses regexps into their constituent sub-expressions that are used to construct NFAs which match the same strings as the given regexps. Placement and routing subroutines were used to construct basic NFA logic building blocks, which converted regexps into NFA logic directly. This was achieved using a one-hot (OHE) technique, which made it possible to construct the NFA logic in O(n) time, where n is the length of the given regexps. The placement and routing subroutines only took O(1) time to construct and process a single character per clock cycle. This made the approach fast and efficient for pattern matching. Cons: The approach needs to be extended further to perform multi-byte and multi-pattern matching.
2. Multi-Character Regexp Matching Designs	
a. Parallel String Matching Engines for NIDS by Tripp (2006).	Pros: Tripp (2006) describes an approach that constructs a FSM which operates on a single byte wide data input. The approach works by assigning a separate FSM for each byte wide data path from a multi-byte input data word. The approach is designed to match the start and end of strings which have a probable chance of occurring part way through a streaming data. A match occurs when all w-byte wide FSM each report a match in a given order at all w instances in the substrings, consisting w-way interleaved search strings. The problem of memory was addressed by Tripp (2008) using a packed-array algorithm that amounts to significant memory savings for larger FSMs. Cons: Although the algorithm avoids the use of two dimensional arrays for storage by using a packed array technique, it is only useful for trivial string patterns.

Table 3.4: (*cont'd*).

Approach	Summary
b. Increased Striding with Run-Length Coding by Brodie, Taylor and Cyron (2006).	Pros: The FSM is built upon the compression technique that improves capacity and speed of matching. The compression technique uses equivalence class identifiers (ECIs) to represent the various compressed inputs in its state transition table. This was achieved by employing directly addressed tables and pairwise combination. The FSM's tables were then compressed and the resulting transition information is deployed in the Indirection and Transition Tables (ITT). An algorithm that consumes multiple symbols in a single clock cycle was implemented to make the design match patterns faster. Cons: The memory efficiency significantly reduced as the number of symbols covered in the final ECIs increased. Also with increased striding came the problem of more memory requirement. Further optimisation was required to reduce the number of accesses to the transition tables.
3. Common Prefix, Infix and Suffix Matching Design by Hieu et al. (2011).	Pros: The scheme involves sharing common prefix, infix, and suffix matching substrings used recognise overlapped matches (Yu et al. 2006). The design further incorporates within it 5 adaptive building logic blocks that make the required FSM for a given set of regexps. The design is able to resolve the problem of false positive matching at the infix block module, by introducing a shift module in each regexp module block. The output of both the infix and shift module blocks are ANDed (logic AND gate circuit), before sending it to the suffix in order to generate a match signal form the prefix sub-pattern. Cons: With character classes involved in the design process, decoding became an overhead. This was replaced with a block RAM centralised character matching (BCCM) scheme. As such, the scheme sacrificed throughput for reduced logic.

Table 3.4: (*cont'd*).

Approach	Summary
4. Shared Character Decoding Design	
a. Extended Shared Decoding Scheme by Clark and Schimmel (2004).	Pros: The scheme implements an approach that sustained a throughput of matching in the ranges of 1 – 100 Gbps. The approach is built upon the one described by Clark and Schimmel (2003), which substituted comparators with decoders during implementation. A higher throughput of matching with lower cost was achieved. The approach by Clark and Schimmel (2004) was able to perform multi-character matching in a parallel arrangement, thereby increasing the speed of matching substrings. Cons: However, the approach incurred higher memory bandwidth and logic overhead. This made it infeasible to scale up the design.
5. Regexp Matching Engine Design	
a. Systematic Translation of Compact REMEs by Yang and Prasanna (2009).	Pros: The scheme translates regexps into compact and high-performance regexps NFA (RE-NFA) matching engines (REMEs). BRAM–based centralised character classifiers were used to match against 1-byte character classes, which made it possible to implement an n-state RE-NFA on a single BRAM. The BRAM has no more than 256 x n bit vector, and uses a two-phased toolchain approach. The first phase gathers together a set of exclusive character classes and assigns to it a floating point sorting key. The second phase sorts the unique character classes based on their sorting keys. The RE-NFAs are then carefully staged and pipelined to perform multi-character matching in a two dimensional arrangement using a priority encoder at all the stages automatically. Cons: The clock frequency of the overall design declines sub-linearly with respect to the state fan-in, but with proper optimisation it could improve further.

Table 3.4: (*cont'd*).

Approach	Summary
6. Classification-Based Designs.	Pros: The classification based designs utilise a simple input compression techniques using the concept of equivalence classification. The technique allows the grouping of all inputs that have the same effect on a given automata. The concept allows the states of a FSM like the NFA to be merged together successfully as described by Ilie, Solis-Oba and Yu (2005). The successfully reduction in the number of transitions and states on the NFA reduces the memory bandwidth and memory requirement. Implementing increased striding (Brodie, Taylor and Cytron (2006) also translate to increase in throughput with lower logic resource overheads leading to higher efficiency. The RFC algorithm by Gupta and McKeown (1999) is capable of assigning proper associated actions to any given rule. The algorithm also decides which flow each arriving packet belongs to. The ECD-NFA approach uses a unique form classification that applies to NFAs only. The approach compresses inputs using an equivalence classification concept. The method is capable of classifying inputs by exploiting the redundancies that normally exists in most transition tables. This is achieved by storing the compressed inputs in a BRAM format that is easily sythesised into a small piece of logic. Cons: The issue of memory bandwidth remains a hindrance to the overall throughput of matching in most approaches. Also, the issue memory utilisation is a problem especially for approaches that are based on non-equivalence classification.

3.3 Chapter Summary

As observed in this chapter, memory centric architectures have proven to be quite crucial in creating minimised automata. Approaches such as the δFA have the ability to reduce number of states and transitions in a given automaton. The δFA requires only a few transitions per state, allowing for faster matching. The approach δFA is an extension of the D^2FA approach, which utilised fast memories such as caches for storing relevant state transition sets. The D^2FA approach sets a diameter-bound for the default transitions of every tree on the overall spanning tree. This improved the memory bandwidth, as more redundant transitions were eliminated. Also equivalent and even non-equivalent state merging is possible by memorising precedent states. Furthermore, with memory reuse, it is possible to reduce the overall memory requirement of the δFA even further.

The reconfigurable and parallel nature of the FPGA architecture could be properly exploited to create highly parallel matching units. The parallel matching units are capable of increasing the speed of

matching regexps. While most of the approaches try to build upon the concept of automatic construction of regexps into NFA logic, others further exploit the idea of using self-reconfigurable hardware devices such as the SRGA. Furthermore, approaches that are centrally focussed on increasing the speed of matching by employing multi-character matching schemes were examined. Such approaches use decoders arranged in parallel, but come at a cost of increased logic resources. The extra costs are mitigated by the use of a centralised BRAM character classification scheme. The BRAM is useful especially by allowing the sharing of inputs among various matching REMEs that are staged and pipelined to perform multi-pattern and multi-character matching. The concept of classification was shown to practically apply to RFC-based algorithms for matching packet headers effectively.

Lastly, equivalence classification based approaches were discussed, which have the tendency to create classified inputs for both DFA and NFA designs. However, the process of creating ECDs is only uniquely applied to the NFA implemented in this thesis. Furthermore, classification based approaches are capable of reducing the overall logic and memory requirement of a given design. If properly designed, such approaches are capable of obtaining higher throughput of matching, with no corresponding increase in the amount of utilised logic circuits. This in turn leads to an improvement in the throughput efficiency of a REME-based design. Improving the throughput and throughput efficiency of NFA REME-based designs is the basis of the study in this thesis and the other reviewed related approaches in general (refer to Section 5.3 for full details).

Chapter 4 analyses the various approaches discussed in this chapter. The aim is to establish by way of comparison, the improvements made by each design to the overall throughput of matching. The remaining approaches that are also concerned with the throughput efficiency of their approach also reported the number of LUTs utilised by their approaches. In Chapter 6, such LUT-based REME approaches were compared with the design implemented in this thesis to ascertain the performance of both the throughput and the throughput efficiency of such designs.

4. Analysis of Related Approaches

This chapter focuses on the various results obtained from the related approaches examined in Chapter 3. The focus is to analyse and compare the approaches with the aim of ascertaining their performance. The analysis is based on the total number of characters consumed, the speed of matching, and throughput among other variables. A single table is also constructed and contains the summary of results obtained from the various sub-sections of the related approaches. The results are then used to analyse the various approaches.

4.1 Tables of Results for Related Approaches

This section shows how the various FPGA-based approaches discussed in Chapter 3 compare to one another based on their design categories. The results to be considered shall be based on the following attributes: the design approach and an n-byte match size, with n = 1, 2 and 4. Also, the speed of matching (MHz), throughput (Gbps), and the total number of characters matched shall be taken into consideration.

The results of each design approach are tabulated and shown by means of simple diagrams. The diagrams are based on the various categories of the related approaches. The entries for each table include: **Design Approach**, which represents the various separate design approaches. The devices used are all FPGA platforms, while **Input** represents the input bus width measured in bytes, and **MHz** represents the design speed. **Tp**, represents the throughput in **Gbps**, while **T/Chars** represents the total number of characters matched. Section 4.1.1 - 4.1.5 contains the table of results obtained for the various approaches discussed in Chapter 3, Section 3.2.

4.1.1 Multi-Character Regexp Matching Designs Table

Table 4.1 shows the design results obtained by Brodie, Taylor and Cytron (2006), Sourdis and Pnevmatikatos (2004) and Yamagaki, Sidhu and Kamiya (2008). The table shows that the approach by Sourdis and Pnevmatikatos (2004) has the best throughput and speed of matching.

Table 4.1: Compared results for multi-character regexp matching designs.

Design Approach	Input	MHz	Tp	T/Chars
Brodie, Taylor and Cytron (2006).	4	133.00	4.26	11126
Sourdis and Pnevmatikatos (2004).	4	303.00	9.71	18032
Yamagaki, Sidhu and Kamiya (2008).	4	113.40	3.63	40896

4.1.2 Common Prefix Sharing Design Table

Table 4.2 shows the design results obtained by Hutchings, Franklin and Carver (2002), Lee et al. (2007), Lin et al. (2006) and Hieu et al. (2011). From the table, the design's speed of matching (MHz) reported by Lee, Hwang, and Park (2007) remained constantly 275.30MHz for a 1-byte and 2-byte character match. However, Table 4.1 shows that the approach by Lee, Hwang, and Park (2007) has the best throughput and speed of matching.

Table 4.2: Compared results for common prefix sharing designs.

Design Approach	Input	MHz	Tp	T/Chars
Hutchings, Franklin and Carver (2002).	1	30.90	0.24	8003
Lee, Hwang, and Park (2007).	2	275.30	4.40	19275
Lin et al. (2006).	1	133.00	1.10	20914
Hieu et al. (2011).	1	231.25	1.85	13287

4.1.3 Shared Character Decoding Design Table

Table 4.3 shows the design results obtained by Clark and Schimmel (2003), Sutton (2004) and Clark and Schimmel (2004). The result for the 4-byte parallel character match reported by Sutton (2004) was considered against each attack patterns. Each result represented the three separate implementations in the design, and the average of each of the three results generated was considered. Also, the results presented by Clark and Schimmel (2004) for a 4-byte match was reported as a useful comparison with the rest of the results as seen in Table 4.3. The table shows that the approach by Sutton (2004) has the best throughput and speed of matching.

Table 4.3: Compared shared/partial decoding designs.

Design Approach	Input	MHz	Tp	T/Chars
Clark C.R and Schimmel E. D (2003).	1	253.00	2.00	17537
Sutton (2004).	4	317.19	10.15	2016
Clark and Schimmel (2004).	4	218.90	7.00	17537

4.1.4 Regexp Matching Engine Designs Table

Table 4.4 shows the results obtained for the approaches by Brodie, Taylor and Cytron (2006), Mitra, Najjar and Bhuyan (2007), Yang, Jiang and Prasanna (2008), and Yang and Prasanna (2009), Ganegedara, Yang and Prasanna (2010), Long et al. (2011) and Singapura et al. (2015). Singapura et al. (2015) reported an 8-byte REME design, with a design speed of 340. 63MHz. As such, the 4-byte match recorded a throughput of about 10.65Gbps only. Meanwhile, Yang and Prasanna (2012) reported two 4-byte match designs. The table shows that the design by Ganegedara, Yang and Prasanna (2010) was able to eliminate all the multiple occurrences of identified rules, using a duplicate checker program. Out of the about 20,000 rules, they realised just about 2,000 distinct rules. This lead to an even greater reduction in the required amount of logic and memory utilised. The average of the respective throughputs and the sum of the characters matched for each pattern matching unit was considered. Also, the average of the results corresponding to the 'webmisc' and 'smtp' rules in the approach reported by Long et al. (2011) were considered. The results are reported in Table 4 and 5 of the reference paper by Long et al. (2011).

Table 4.4: Compared results for regexp matching engine designs.

Design Approach	Input	MHz	Tp	T/Chars
Brodie, Taylor and Cytron (2006).	4	133.00	4.26	11126
Mitra, Najjar and Bhuyan (2007).	16	100.78	0.81	10977
Yang, Jiang and Prasanna (2008).	4	233.13	7.46	15000
Yang and Prasanna, (2009).	4	300.00	9.60	28000
Yang and Prasanna (2012).	4	198.6	6.36	120000
Yang and Prasanna (2012).	4	166.7	5.33	100000
Ganegedara, Yang and Prasanna (2010).	4	202.90	6.50	16384
Long et al. (2011).	1	155.50	1.24	1020
Singapura et al. (2015).	8	340.63	21.8	100000

4.1.5 Classification-Based Designs Table

From the discussion in Section 3.2.2f, the results obtained for the approach by Brodie, Taylor and Cytron (2006) were considered. The full analysis is performed together with that of the approach implemented in this thesis and other REME related designs in Chapter 6. Not many approaches exist on regexp matching using classification-based approach to regexp matching. The scheme by Brodie and Cytron and Taylor (2006) was designed for DFA's and not NFAs. However, the table form of input compression described in the scheme was suitable for increasing the stride of any well-constructed NFA design. The approach by Gupta and McKeown (1999) reported no results. However, their hierarchical form of input classification was also useful in the design described in Chapter 5.

Table 4.5: Design result for an ECI-based design.

Design Approach	Input	MHz	Tp	T/Chars
Brodie, Taylor and Cytron (2006).	4	133.00	4.26	11126

Table 4.6 comprises of all the 18 separate results reported for the various related approaches under the various categories of designs. The results are as presented in Table 4.1 – Table 4.5.

Table 4.6: Combined table results for the various FPGA-based approaches.

Design Approach	Input	MHz	Tp	T/Chars
Brodie, Taylor and Cytron (2006).	4	133.00	4.26	11126
Sourdis and Pnevmatikatos (2004).	4	303.00	9.71	18032
Yamagaki, Sidhu and Kamiya (2008).	4	113.40	3.63	40896
Hutchings, Franklin and Carver (2002).	1	30.90	0.24	8003
Lee, Hwang, and Park (2007).	2	275.30	4.40	19275
Lin et al. (2006).	1	133.00	1.10	20914
Hieu et al. (2011).	1	231.25	1.85	13287
Clark C.R and Schimmel E. D (2003).	1	253.00	2.00	17537
Sutton (2004).	4	317.19	10.15	2016
Clark and Schimmel (2004).	4	218.90	7.00	17537
Mitra, Najjar and Bhuyan (2007).	16	100.78	0.81	10977
Yang, Jiang and Prasanna (2008).	4	233.13	7.46	15000
Yang and Prasanna, (2009)	4	300.00	9.60	28000
Yang and Prasanna (2012).	4	198.6	6.36	120000
Yang and Prasanna (2012).	4	166.7	5.33	100000
Ganegedara, Yang and Prasanna (2010).	4	202.90	6.50	16384
Long et al. (2011).	1	155.50	1.24	1020
Singapura et al. (2015).	8	340.63	21.8	100000

4.2 Analysis of Related Designs

The analysis of the results as seen in Table 4.6 is based on the: speed of matching (MHz), the number of bytes consumed at once, the total number of characters matched, and the throughput (Gbps) only. This is because basic concern of this thesis like the other approaches is focused on improving the throughput of matching.

4.2.1 Data Analysis of Results

a. Single-Data Graphs

Figure 4.1 - 4.4 shows the graphs of the results reported by the various designs as they appear in Table 4.6. From Figure 4.1, it can be seen that about 77.78% of the designs implemented a multi-character matching design, with about 55.56% of them using a 4-byte character matching scheme. This was done with the aim of improving the overall throughput, which is often at the expense of incurring more logic circuit cost. However, the 16-byte input attributed to the design by Mitra, Najjar and Bhuyan (2007) and as shown in Figure 4.1 is split into sixteen 1-byte matching units. The sixteen matching units are combined into a single cohesive matching engine that outputs 16-bits. However, all of the combined matching units produced a cumulative throughput of 12.90 Gbps, which was split among the separate matching units.

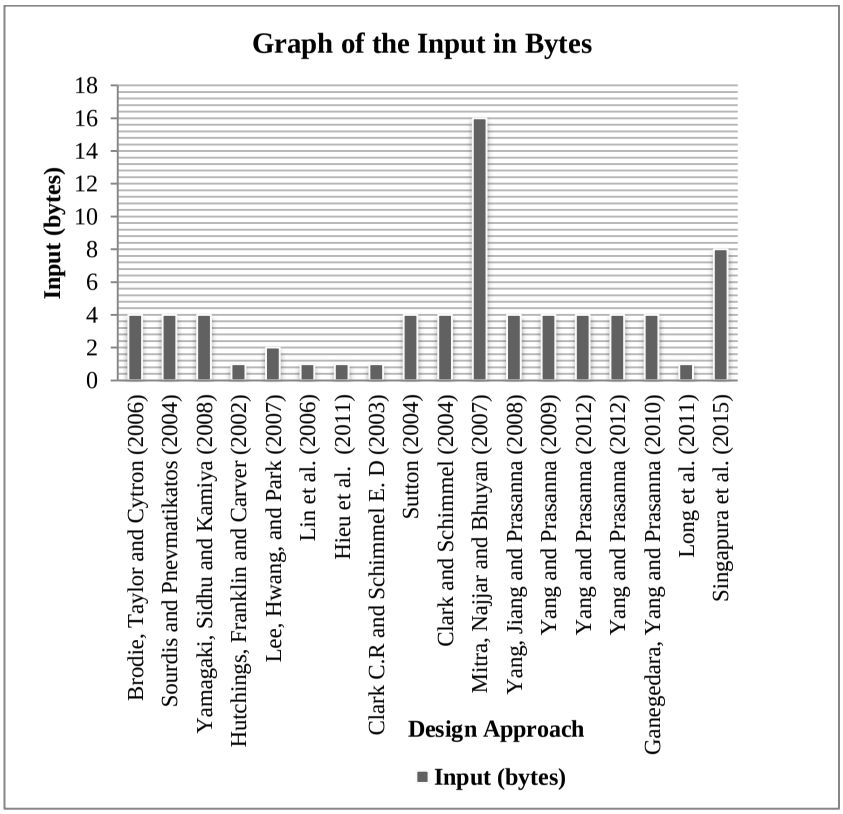


Figure 4.1: Graph of the input (n-bytes, n = 1, 2, 4, and 16).

Figure 4.2 shows that 77.78% of the total characters matched per design are between the ranges of 10,000 – 20,000 characters, for even the most efficient implementations. It also implies that going beyond that range does not necessarily translate to higher speed or higher throughput of matching for any 4-byte matching REME design. Further discussion on that is found in Section 4.2.1b concerning Figure 4.5.

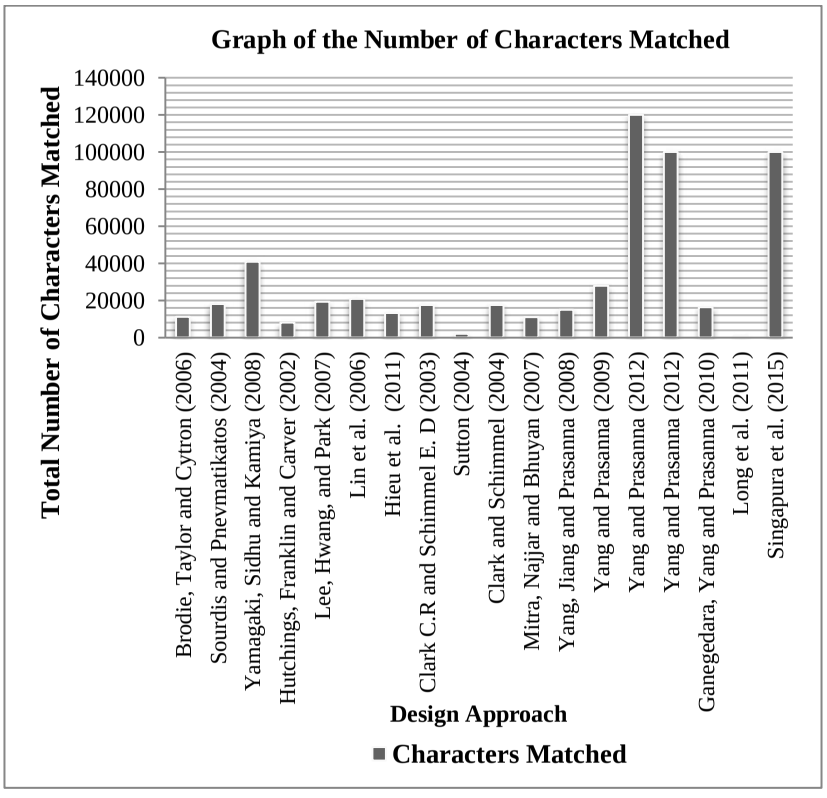


Figure 4.2: Graph of the number of characters matched (ranging from 652-120000).

Figure 4.3 shows that the average throughput obtained by about 55.56% of the designs is around 4Gbps. About 33.33% of them fell between the ranges of 7.5Gbps - 10Gbps. Those designs having 4Gbps throughput have designs that clocked at speeds below 150MHz, while those between the ranges of 7Gbps – 10.65Gbps obtained speeds between the 200MHz – 340MHz range.

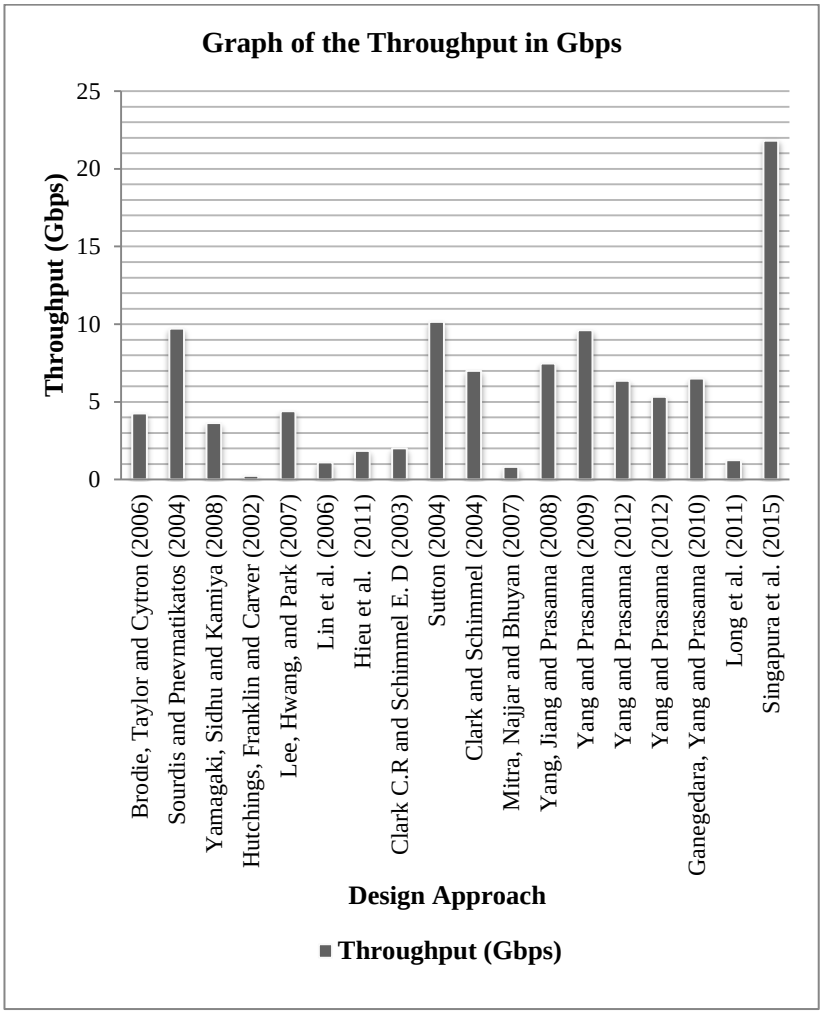


Figure 4.3: Graph of the throughput (ranging from 0.24-10.65 Gbps).

Figure 4.4 shows that about 72.22% of the designs could not obtain beyond the speed range of 270MHz – 280MHz. Any speed beyond that range would reflect an efficiently constructed design, which is rare especially beyond the 300MHz mark. However, most modern FPGAs such as the Xilinx FPGA Virtex-7 device clock at 500MHz. Also, only a very compact and efficient design can take full advantage of the clock speed and the large amount of parallel logic and on-chip memory that is now available on such a device.

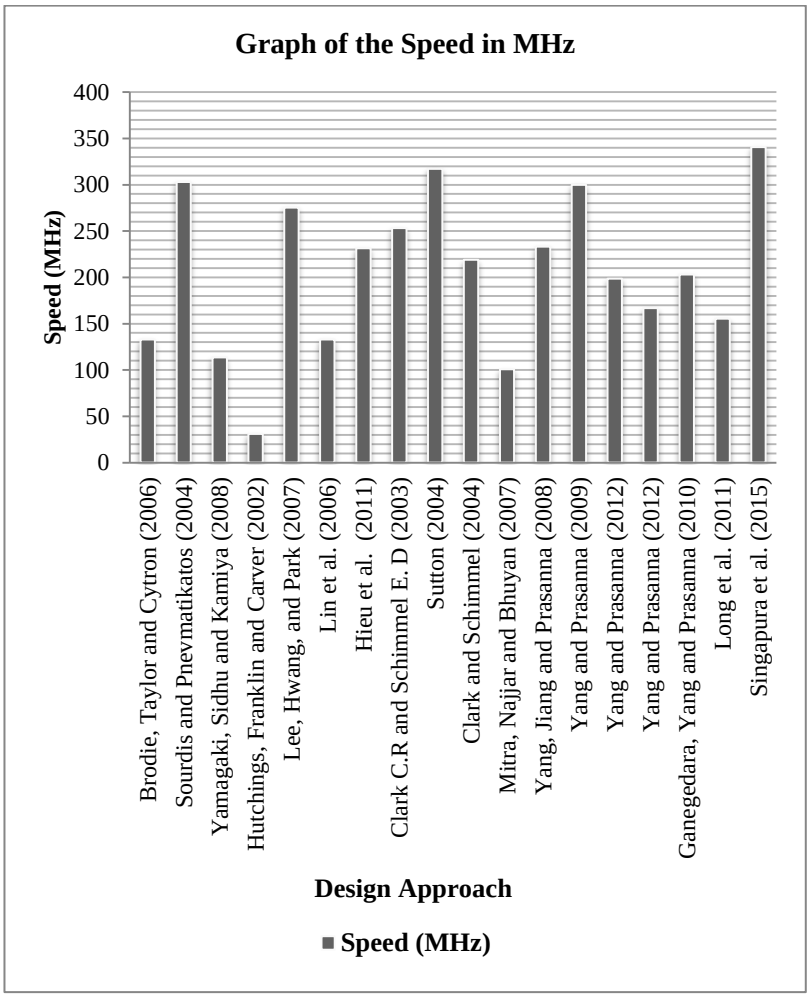


Figure 4.4: Graph of the speed (ranging from 30.90MHz – 340.63).

b. Double-Data Graphs

Figure 4.5 shows the graphs which are based on the results shown in Table 4.6. The graphs are used to compare the relationship between the throughput and speed of matching, as well as the throughput and the total number of characters matched by each design. Looking at this perspective and starting with Figure 4.5, it can be seen that increasing the number of characters beyond the ranges of 10,000 - 20,000 characters does not necessarily improve the throughput.

The design by Yang and Prasanna (2012) reported two separate 4-byte matching design implementations, while Singapura et al. (2015) reported the results of an 8-byte matching design. The two separate implementations by Yang and Prasanna (2012) recorded throughputs of 6.36Gbps and 5.33Gbps, while Singapura et al. (2015) recorded a throughput of 21.8Gbps. However, the 8-byte matching design by Singapura et al. (2015) recorded a design speed of 340.63 MHz. As such, for the purpose of comparison with all the other 4-byte matching designs, the throughput in the design by Singapura et al. (2015) is about 10.65Gbps for an equivalent 4-byte matching design. Figure 4.6 indicates that the higher the speed, the higher the throughput of matching. Obtaining an increase in design speed can only be possible when a design is further optimised and made more compact.

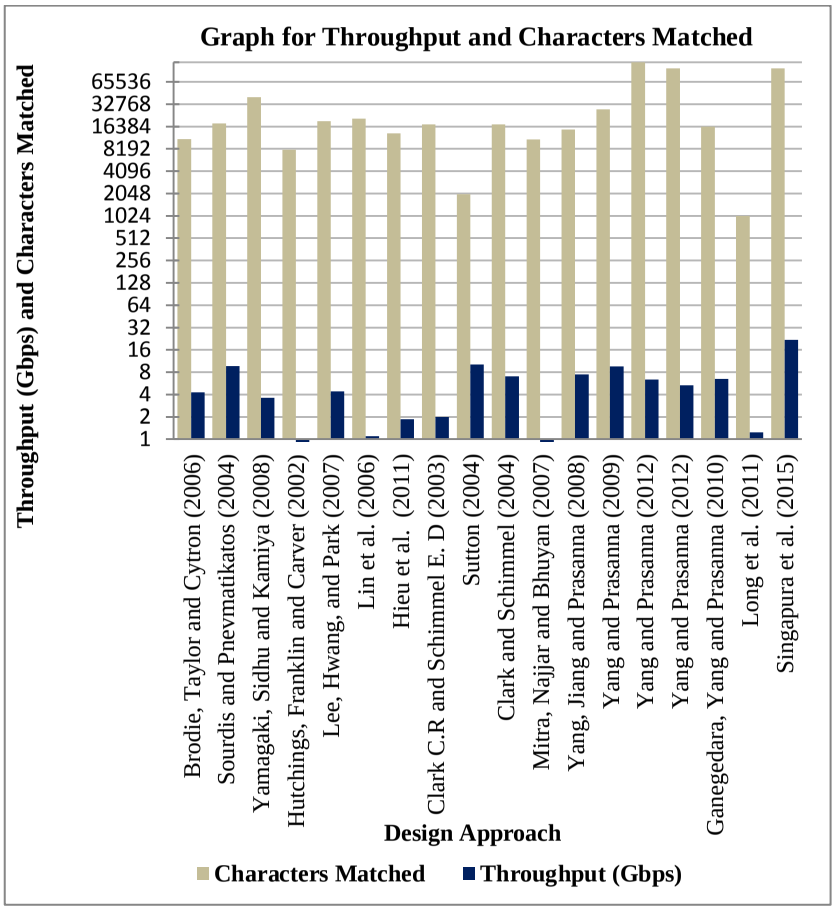


Figure 4.5: Distribution of the throughput and the total number of characters matched.

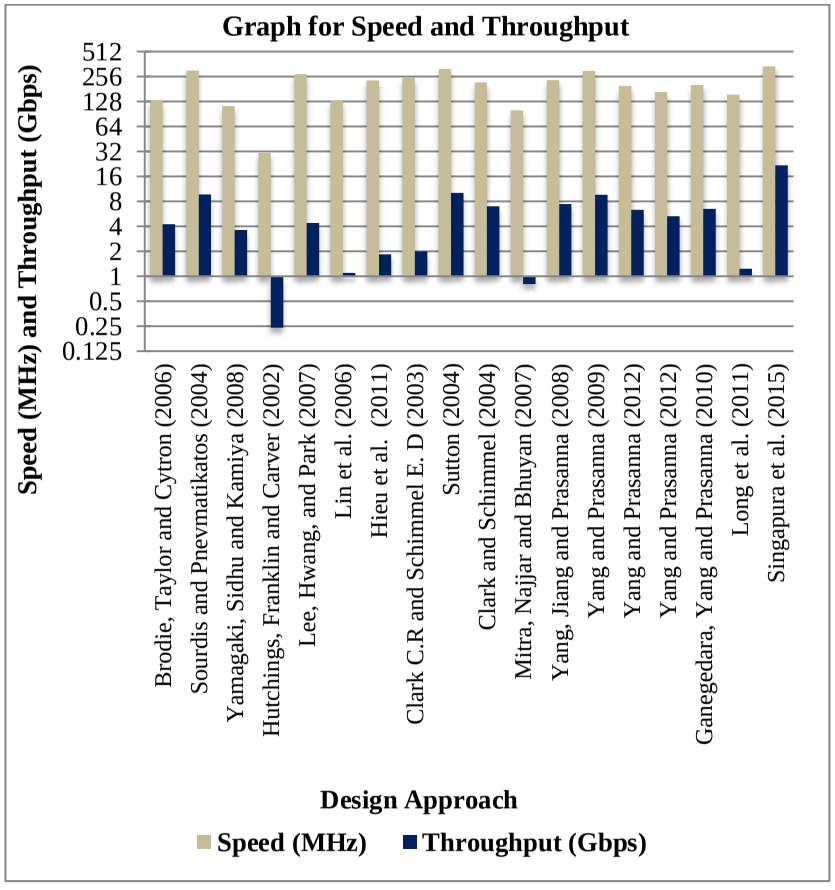


Figure 4.6: Distribution of speed and the throughput of matching.

4.3 Chapter Summary

The various FPGA-based designs are multi-character regexp matching designs. Some of the designs reported results for multi-character matches. However, most of the multi-character matching designs that are considered in most of the related approaches are 4-byte matching designs, and is considered a standard. The various designs that consume more than 4-bytes of characters only show slight improvement in the throughput and not in the overall design clock speed. This is because consuming more than four characters per clock cycle increases the number of memory accesses.

Furthermore, the higher the memory latency the more the decline in the overall design operational clock frequency. As a result, a good 4-byte matching system combined with a better optimisation strategy is seen to produce better results. Another observation is that if a design combines multi-character and multi-pattern matching in the same design engine, the clock speed and the ratio of utilised logic further declines. However, the design in this thesis combined both multi-character and multi-pattern matching into all the four separate design engines. The full description of the design is found in Chapter 5 and later analysed in Chapter 6.

5. **Design and Implementation**

This chapter describes the FPGA-based approach that uses the concept of equivalence classification to implement a set of efficient parallel regexp pattern matching engines (REMEs). The description of the approach starts with the design called ECD-NFA, leading up to the optimised version called ECD_RTS-NFA. The design was first introduced in Sections 1.4 and 1.7, and is fully discussed in this chapter.

5.1 Motivation

This thesis is based on the desire to build an efficient REME-based design. The design takes advantage of the fine-grained parallelism provided by FPGAs. Current rules contain complex regexps, which when implemented place high demand on memory. The complexity of the current rules makes software regexp search engines slow and non-scalable.

However, the main motivation behind the design approach implemented in this thesis is to significantly reduce the storage and processing time attributed to most NFA-based designs and their variants (Sidhu and Prasanna 2001). An FPGA is known to have good clock rate (Brodie, Taylor and Cytron 2006) which enhances performance. As such, the major challenge was how to come up with a fast and more scalable hardware regexp pattern matching approach. The approach should obtain higher throughput and lower logic circuit costs, compared to all the other related approaches. The following summarises the motivations for this thesis:

i. There is the need to create a very simple and less complex toolchain (Yang and Prasanna 2009) for implementing simple, fast and efficient logic circuit NFA-based REMEs. The idea is to create a new design that outperforms the ones implemented by Mitra, Najjar and Bhuyan (2007), Yang, Jiang and Prasanna (2008), Yang and Prasanna (2009), and Ganegedara, Yang and Prasanna (2010). The related approaches mentioned are more directly concerned with the type of parallel REMEs implemented in this thesis. Furthermore, the design is split into two phases namely: the first phase (Software implementation) and the second phase (Hardware implementation on FPGAs).

ii. The need to effectively utilise techniques that combine optimisations such as: edge reduction, alphabet reduction, increased striding (Becchi and Crowley 2008, p. 50), input classification (Brodie, Taylor and Cytron 2006; Tripp 2006; Arnold 2007), and prefix, infix and prefix sharing (Hutchings, Franklin and Carver 2002; Sourdis and Pnevmatikatos 2004; Yu et al. 2006; Lee et

al. 2007; Lin et al. 2006) into a single approach. Section 3.2.2c discussed the various optimisation techniques.

iii. The need to create a design that performs multi-character and multi-pattern matching. This is in relation to the approach implemented by Sourdis and Pnevmatikatos (2004), Becchi and Crowley (2008), Clark and Schimmel (2004), Jiang, Yang and Prasanna (2010) and Yang and Prasanna (2012). The design also needs to be fast and capable of making the optimal use of the limited available FPGA logic resources.

iv. To construct a design that builds into each REME in (i), multi-NFA REME blocks. The REME blocks are to be arranged in parallel configuration. The arrangement will allow for multi-character and multi-pattern matching, while utilising fewer REMEs than those constructed by Mitra, Najjar and Bhuyan (2007), Yang, Jiang and Prasanna (2008), Yang and Prasanna (2009), and Ganegedara, Yang and Prasanna (2010). The design shall utilise the unique form of table-synthesis of ECDs generated in the first phase of (i).

v. Lastly, there is the need for the design to automatically and efficiently generate and integrate the required VHDL files, which are the expected outputs of the first software implementation phase. The generated output files are then supplied to the hardware in the second phase of the design toolchain for implementation.

vi. The ECDs (refer to Section 1.4) shall be used to represent state vectors in the transition table of the ECD-NFAs. The inputs have the same effect on the automata, and exploit any redundancies normally associated with table-driven automata. The ECDs shall be generated in the first phase of the design, which will later prove to be useful when combining multiple NFAs into a single composite NFA. Furthermore, each time 2^i-byte ECDs are generated recursively, where i = 1, 2,..., higher byte input are generated, and required to perform multi-character matching. The ECDs grow steadily at O(k), where k is the number of ECDs created. This is achieved by performing a logical cross product computation of two 1-byte ECDs state vectors. The process ensures that for a 4-byte match process to occur for instance, n will not grow beyond the value of 127. Any value where $n > 127$, will consume too much logic e.g. LUTs and registers during the synthesis and implementation stage. As such, the higher the amount of components consumed, the poorer the obtained throughput efficiency (Yang, Jiang and Prasanna, 2008).

5.2 Design Concerns

Common prefix sharing is a basic feature considered in the design described in this chapter. The feature has been implemented by many approaches such as the ones by Hutchings, Franklin and Carver (2002), Lee et al. (2007), Lin et al. (2006), and Hieu et al. (2011). Prefix sharing is used to reduce the amount of logic and regexp matching engines used to represent patterns with common prefixes. This is achieved by sharing common input strings. Hutchings, Franklin and Carver (2002) succeeded in accelerating string matching by implementing common prefix matching.

The approach also extended the NFA-based hardware implementation strategy described by Sidhu and Prasanna (2001) to include an additional operation called optionality. The operation is used to match zero or one input character per clock cycle. However, Hutchings, Franklin and Carver (2002) admitted that their design was far from being a complete solution, but hoped to address it in future. However, circuit optimisation is never an issue in the optimised ECD_RTS-NFA design version. This is because the

ECD_RTS-NFA handles optionality much more efficiently. However, circuit optimisation remained an open issue for Hutchings, Franklin and Carver (2002).

Shared decoding approach (Clark and Schimmel 2003; Sutton 2004; Clark and Schimmel 2004) is also an important feature used in this thesis. There was the need to technically minimise the routing wires and logic circuits required for accessing character inputs by avoiding the use of character comparators. The comparator approach (Sourdis and Pnevmatikatos 2003) often failed to deal effectively with the issue of poor design speed and increased logic size. This is because the more characters are heavily pipelined, the more the clock latency between the FFs, which is attributed to the distance between them.

However, even with a decoding approach constructed to perform multi-character matching, the overall design speed degrades significantly. This is because, the larger the design, the more the fan-out of the input stream required to traverse longer distances. A solution was obtained by using the concept of classification to reduce the size of the design. Using the classification technique, the design input bit size was reduced to 7-bits inputs instead of 8-bit inputs. The ECD-NFA design only utilised 7-to-128 decoders originally before it was optimised. This is a variation from the 8-to-256 decoders utilised in the approaches described by Clark and Schimmel (2003), Sutton (2004) and Clark and Schimmel (2004).

Furthermore, the use of 7-to-128 decoders helped to save about 50% of the redundant output wires compared to the approaches by Clark and Schimmel (2004). Without the reduction in the size of the decoders, a high logic circuit cost is incurred when integrating it into a multi-character and multi-pattern matching design such as the one implemented in this thesis. However, even with such reductions, the ECD-NFA design still consumed too much logic resource. The design was too slow to synthesise and non-scalable too. As such, the design was optimised and the new version was called ECD_RTS-NFA (refer to Section 5.3.3 for more details).

The ECD-NFA approach is suitable for utilising fewer 6-bits input LUTs, registers and other circuit components, contained within each respective CLB. The utilised CLBs are closely packed together due to the nature of the design forcing the circuits into a confined region of the FPGA during circuit optimisation. Such efficient space utilisation helped to cut down the overall clock and routing delays experienced when performing parallel regexp matching. The speed of matching, the throughput, and throughput efficiency (Yang, Jiang and Prasanna 2008) improved in the process. Also, during the construction phases of the design implemented in this thesis: (a) regexps with long repetitions were avoided (Yang, Jiang and Prasanna 2008) such as "…[^\n]{1024}", but allowed regexps with flags: 's', 'm', 'i' (Long et al. 2011, p. 70), and (b) selected patterns with average lengths too (Yang, Jiang and Prasanna 2008).

The design by Tripp (2006, p. 26) which was based on a memory saving scheme constructed for a difference array, is capable of storing all the table entries that are different from the values of the given default array. The array is then decomposed into a sequence of state vectors. The method that put together the state vectors into a one–dimensional packed array was later implemented by Tripp (2008, p. 4). The aim was to avoid clashes among the various active entries on the table. A similar idea for generating a one-dimensional vector of output bits by Tripp (2008) was also utilised in the table-synthesis module of the ECD_RTS-NFA REME designs for generating the outputs used to supply ECDs to the NFA block modules.

5.3 The Hardware Design Phase

The block design of the original ECD-NFA approach is divided into two phases as shown in the block diagram of Figure 3.26. The block diagram as shown in Figure 5.2 summarises the flow of the entire scheme. The first phase is made up of blocks labelled: Snort rules, PCRE regexps file extractor, parser, constructor and optimiser is connected to units (a), (b), and (c) (refer to Section 5.4.1 for details). The second phase is made up of the block labelled as hardware ECD-NFA/ECD_RTS-NFA (FPGA) builder. Section 5.3.1 explains each of the decomposed modular components as shown in Figure 5.2.

5.3.1 The Modular Block Module: First Phase

a. Snort Rules/PCRE Extraction and Parsing Phase

The Snort NIDS (Snort 2013; Roesch 1999, p. 229) was explained in Section 2.2.1 and 2.4. The regexps that are automatically extracted from the rich collection of regexps found in Snort rule database. Afterwards, the regexps are then read into the parser for subsequent conversion into tokens by the tokeniser module of the parser. The tokens are then used to construct the required parse trees used to build the various original ECD-NFAs. The parser, as shown in Figure 5.2 is based on the software initially implemented by Tripp (2008) for a DFA implementation. It is made up of various newly added and existing modules which have been updated accordingly for the NFA implemented in this thesis.

Another module that was added was the regexp extractor module made up of the first two sub-modules as shown in Figures 3.26 and 5.2 namely: PCRE regexp files generator. The extracted files serve as inputs into the parser module. The parser module then integrates the files automatically as described in Section 5.3.1b, and illustrated in Figure 5.2. The regexp extractor is capable of extracting four n-regexp files at once, where n=2, 3, 4, and 5 using a random function. The function is designed to select regexps without introducing bias by avoiding all manual processes of selection. This is because if the selection is biased or manually done, there is a chance that regexps with large constrained repetitions e.g. a{1024}will be intentionally avoided. That way only trivial regexps with shorter lengths may end up been utilised and that cannot prove the efficiency of the design, especially in a worst-case scenario. As such, the regexp extractor often ends up selecting extremely complex regexps which explains the fluctuations in the design results (refer to Section 6.2 of Chapter 6 for result details). Furthermore the regexps are extracted directly from the VRT Rule distributed by Sourcefire (v 2.0) community rules, 2001-2013 (Snort 2013; Sourcefire 2009) provided by Snort. The files are afterwards uploaded and processed by the ECD-NFA parser.

b. The ECD-NFA Design

The construction and optimisation block as shown in Figure 5.2 is responsible for building the basic ECD-NFA building blocks. The blocks are then modified to construct the multi-character and multi-pattern matching NFA REMEs. The REMEs are generated, integrated and transferred for subsequent hardware implementation. The unit labelled (a) in Figure 5.2 is the unit responsible for generating the BRAM equivalent of the hardware memory. The module is used to compress and store the table of generated ECDs.

The unit labelled (b) in Figure 5.2 represents the module responsible for generating the table of compressed n-byte matching ECDs, where n = 2 and 4. The tables are later synthesised into logic during the hardware processing phase to perform table look up operations used for multi-character matching. The

process is iterated by scaling up the number of regexps per engine. The process also ensures that the number of ECDs generated per iteration for each combination of regexps, does not exceed 128 ECDs (0 - 127) as explained in Section 5.1-vi. Figure 5.1 illustrates an n-byte multi-character and multi-pattern matching ECD-NFA engine. Looking at Figure 5.1, the label C represents the compression process in parser, while BRAM is a 4x256x8-bits memory block of compressed 1-byte ECDs. The label T_c represents a compressed-2D (two-dimensional) table of 2-byte and 4-byte ECDs. The label ≤5-regexps represents 4xECD-NFAs created for matching up to 5 regexps. Figure 5.1 is fully elaborated in Section 5.4. Furthermore, 4-bytes of characters are streamed into the block engine per clock cycle. The streamed characters are then compressed into 2-byte and 4-byte ECDs in the n-byte ECD class table constructor labelled (b), where n = 2 and 4, as shown in Figure 5.2.

The process is recursively performed to take on two copies of the 2-byte ECDs table and used to generate 4-byte ECD tables of compressed inputs (refer to Section 5.4.1a for more details). Meanwhile, the BRAM module in the parser is used to store the compressed 1-byte table of ECDs. The BRAMs provide a method of compression from raw data inputs to the various ECDs. The raw data inputs are then mapped by the BRAM module to their respective ECDs. Label (a) in Figure 5.2 contains the BRAM module responsible for the memory compression of the ECDs. When the construction process of the first phase is completed, the BRAM module is converted to an equivalent VHDL file representation of the actual 256x8-bits BRAM component on the hardware.

The four ECD-NFAs corresponding to each 256x8-bit BRAMs and decoding modules are generated, as shown in Figure 3.26 and Figure 5.2. Each of the ECD-NFAs is driven by the inputs from the 4-byte table of compressed ECDs generated in label (a) of Figure 5.2 and as 4 x T_c in Figure 5.1. Four of the sub-REME blocks as shown in Figure 5.1 make up a single REME block. This forms the typical arrangement found in every REME block as shown in Figure 5.3 and Figure 5.7. The table look up operation is performed in the component labelled T_c in Figure 5.1 and the decoding units of Figure 3.26. The outcome is an n-bit vector of outputs used to supply ECDs to the four sub-ECD-NFAs, where n = 2 and 4. The ECD-NFAs then perform the matching of the various ECDs with each of the four sub-ECD-NFAs releasing 1-bit of output, which is encoded to form a 4-bit output vector.

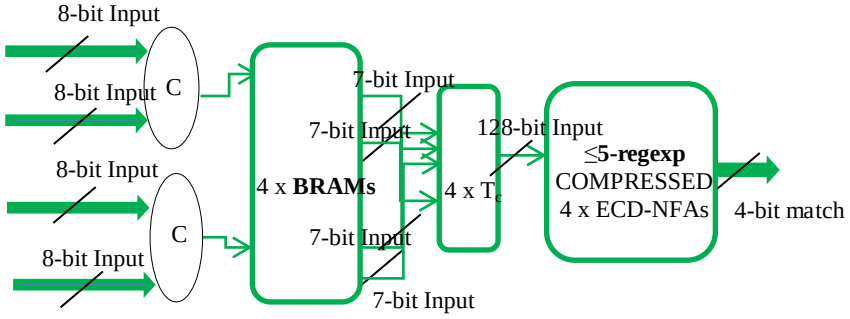


Figure 5.1: A 4-byte ECD-NFA for an n-regexp engine, where n = 2, 3, 4, and 5.

c. VHDL Generator

Once the design phase described in Section 5.3.1b is completed, the three units labelled (a), (b) and (c) in Figure 5.2 are then combined into multi-character and multi-pattern NFA blocks. The blocks are then laid out and saved as VHDL files. The files are made ready for integration and subsequent synthesis in the XST VHDL synthesis tool, during the hardware implementation process in the second phase.

This stage is quite crucial to the design in this thesis, because it saves so much construction time and resource. It also optimises the entire automata before generating the VHDL files. Attempting to achieve the same strategy through direct construction and optimisation on the hardware, would have been inefficient. The entire process would have been manual, difficult, error prone and time consuming, even when carrying out unit or modular testing. It is also more flexible to effect changes and updates to the design which is easily re-compiled afterwards. This is because the actual logic circuit implementation is not required at that point. With the VHDL files generated, the second phase of processing begins in Section 5.3.2.

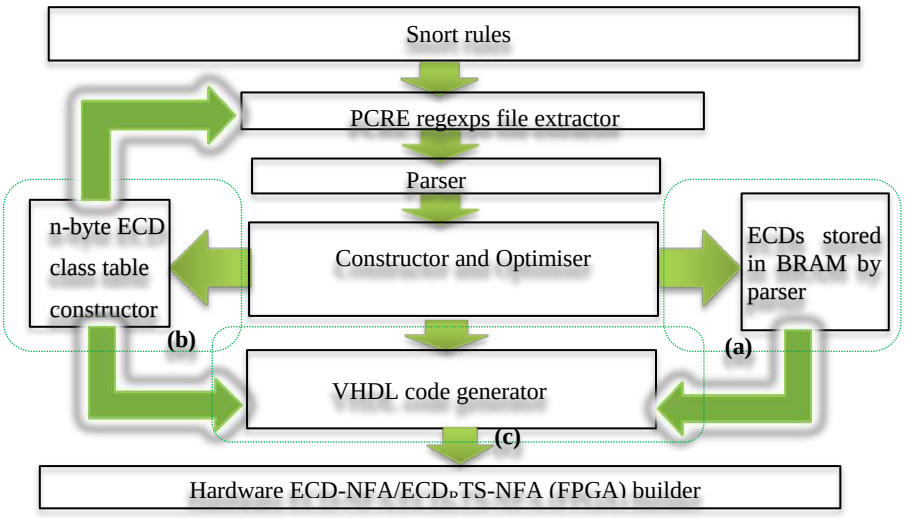


Figure 5.2: Software/hardware ECD-NFA/ECDRTS-NFA model.

5.3.2 The Hardware Module: Second Phase

a. Hardware Design

This section deals with the hardware implementation of the design in this thesis, using the XST VHDL synthesis tool. It is the last block in Figure 5.2 labelled as ECD-NFA/ECD$_R$TS-NFA (FPGA), and the design description is elaborated further in Section 5.4. The block is partitioned into a number of modules. The block module is pieced together with four sub-ECD-NFA blocks, four sub-table/decoding blocks and four sub-BRAMs blocks. Each REME block is made up of a memory component, a table/decoding component and an NFA component as shown in Figure 5.3. The block modules are automatically integrated into the VHDL file as the outputs of the software phase. The table/decoding module performs the function of decoding the respective inputs fetched from the memory component as shown in Figure 5.3. The module uses the decoder circuit components for its decoding operations.

Each of the separate engine blocks as shown in Figure 5.3 is made up of an average of 20 regexps or less. Each of the engines is as shown in Figure 5.1. However, the problem with the table/decoding module of Figure 5.3 is that it is time consuming and involves nested loop operations performed on four 7-bit input decoders. FPGA synthesis tools are not designed to process such complex operations, which made it near impossible to synthesise the design even when trivial regexps were considered. It became clear that a major modification needed to be made to the design to eliminate nested loop operations. The decision led to the creation of the optimised version of the ECD-NFA called ECD$_R$TS-NFA (refer to Section 5.3.3).

Furthermore, each row of the parallel REME blocks as shown in Figure 5.3 has four 256x8-bit BRAMs modules storing the compressed 1-byte ECDs as explained in Section 5.3.1b. Each of the four 256x8-bit memories outputs 7-bits of ECDs. The ECDs are then concatenated to form 28-bit outputs. The output is then supplied as input to the table/decoding component to perform table look up operations. Once the operation is performed, a vector of < 128 bits is released as output. The output represents the <128 ECDs supplied as inputs to the block of four sub-ECD-NFAs. This is achieved by supplying a group of 7-bit ECDs to each one of the four sub-ECD-NFAs simultaneously. As such, it is possible to match up to 20 regexps if each sub-ECD-NFA was constructed with up to 5 regexps per block (refer to Section 5.4 for details).

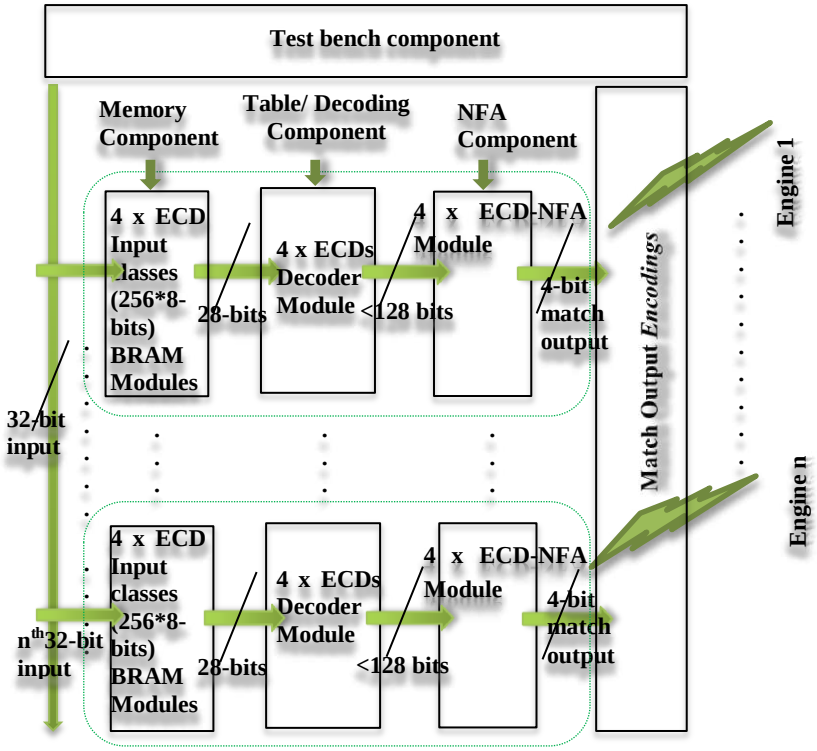


Figure 5.3: Block diagram of parallel 4-byte ECD-NFA REMEs.

The remaining modules in this phase include the user constraint file (UCF) module for setting the clock timing, and the test bench module which specifies in VHDL the complete simulation environment for the analysed system unit under test (UUT). The test bench module contains both the UUT as well as the stimuli (data) for the simulation. Upon the completion of the design, the top level module is set accordingly. Afterwards, the synthesis process of compilation begins, in order to realise the design. Section 5.4 explains the various stages which the implementation of the design has to pass through in order to get it ready for transfer into the target Xilinx FPGA Virtex-6 device. The design as shown in Figure 5.3 is further elaborated in Section 5.4.

5.3.3 The ECD$_R$TS-NFA Design

The optimised version of the ECD-NFA design referred to as the ECD$_R$TS-NFA retains the basic constructs of the former as described in the first phase of the design construction phase. The optimisation process particularly affects the middle table/decoding component block module of each REME. The major change made to the block module involved the total eradication of the decoders used in the module and the operations involved. Also, the shift registers, LUTs and other logic components used by the

decoding units were eradicated too. The optimised block component now called the table-synthesis block simply synthesises the table of ECDs into pure logic. The logic components are a few LUT RAMs, shift registers and other logic elements needed to perform the given look up operations. The operations became almost a linear process compared to the initial nested loop operations performed with the decoders and shit registers in the initial design. The direct consequence of these optimisations led to increased speed of matching and reduced synthesis time of the overall design. The process of LUT packing became more efficient.

The optimised design only utilises 2x36kBRAMs dual-port primitive memories containing each of the 4 tables of 1-byte ECDs described in Section 5.3.2a. This reduction translated to just four 256x8-bit BRAMs utilised for every 20 regexps REME blocks in the ECD_RTS-NFA design. This is opposed to the initial five 256x8-bit BRAMs for every 5-regexp or less REME block in the previous ECD-NFA design. This translated to a significant reduction in the number of required primitive BRAMs, from 20x36kBRAMS to just 2x36kBRAMs for every 20 regexps REME blocks. As such, the design was able to save about 90% of the memories required from original ECD-NFA design. The detail of the optimised approach is also elaborated in Section 5.4. From now henceforth, discussion will dwell on the newly optimised ECD_RTS-NFA design.

5.4 The Implementation Phase

5.4.1 Implementation of the First Phase

This section elaborates more on how each of the aspects mentioned in Section 5.3 (design phase) together with the various modules and units as shown in Figures 5.1 - 5.3 are related. The discussion begins with the implementation of the first phase of the design.

a. The ECDs and ECD_RTS-NFA Algorithm

The classification approach, like the one implemented by Yang, Jiang and Prasanna (2008), does not differentiate between individual, ranges or general character sets. This is because they are all referred to as character classification approaches (Yang, Jiang and Prasanna 2008). The approach consumes one character as input, and generates a match bit. The match is represented by the bit position it occupies on the output match vector, as illustrated in Algorithm 5.1. Algorithm 5.1 is used for constructing the ECD_RTS-NFA design. The algorithm is also used to generate the required ECDs assigned to each of the respective state vectors and to demonstrate how the implemented design in this thesis actually works in practice. Given a simple example of a regexps such as: "/(a|b)*cd/", the ECDs representing the state vectors based on Algorithm 5.1 was obtained. The equivalent state table and NFA are then generated. Based on the regexp "/(a|b)*(cd)/", and the generated ECDs used to represent the various vectors of next states as seen in Table 5.1, the equivalent ECD_RTS-NFA is first constructed. Afterwards, the NFA is minimised to create a more compact ECD_RTS-NFA as shown in Figures 5.4 – 5.6. Figure 5.6 shows the constructed ECD-NFA with ECDs assigned as the new inputs for the transitions.

It is interesting to know that all the NFAs as shown in Figures 5.4 – 5.6 perform the same matching process, but in different reduced forms. The total number of states and transitions on the original NFA as shown in Figure 5.4 has now been reduced from 11 states to 5 states as shown in Figure 5.5. Also, the number of transitions has reduced from 13 transitions to as little as 7 transitions. All the epsilon and self-transitions in Figure 5.4 have been eliminated as shown in Figure 5.5. The letter z as seen

in Figure 5.5 represents all those characters in the ASCII character set that are not letter a,b,c, or d (z is simply represented by the class of characters [^abcd]). This reduction shows that while minimising the original NFA to form the ECD_RTS-NFA, about 45% of the states and 53% of the transitions in the original NFA have been eliminated. This further created a more compressed and compact ECD_RTS-NFA as shown in Figure 5.5 and Table 5.1 (which shows how the ECD_RTS-NFA works). In Table 5.1 ECD 0 represents the input class [ab], ECD 1 represents input c, ECD 2 represents input d, and ECD 3 represents the input class [^abcd].

On the input of ECD (0-3) as shown in Figure 5.6 and Table 5.1, the NFA remains on the initial state 0. On the input of ECD 0 on state 1, the NFA transits from state 1 to states 1 and 2. On the input of ECD 1 on state 2, the NFA transits from state 2 to state 3. On the input of ECD 2 on state 3, the NFA transits from state 3 to state 4. State 4 is the accepting state.

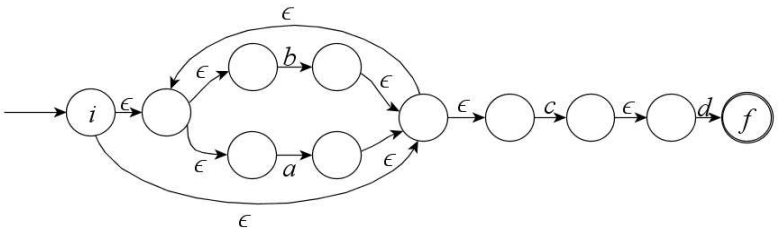


Figure 5.4: NFA for the regexp (a|b)*(cd) (Sidhu and Prasanna 2001).

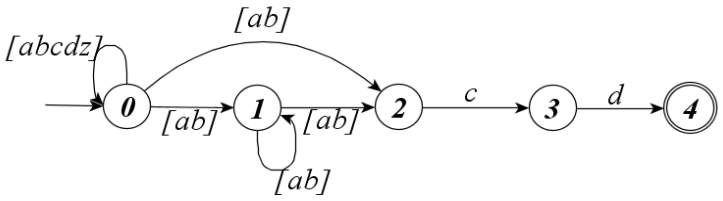


Figure 5.5: Minimised classified ECDRTS-NFA.

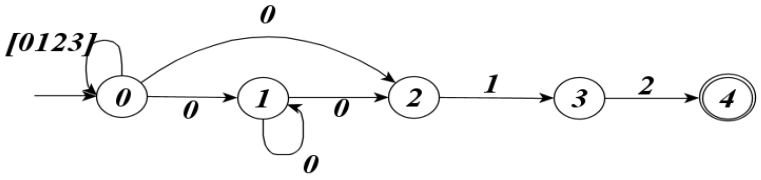


Figure 5.6: ECD-NFA with assigned ECDs.

By performing steps (i) to (v) in Algorithm 5.1, the required table of compressed ECDs is then generated for the 2-byte match inputs. This is achieved by taking the cross product of each ECD column against itself and against any other ECD column as seen in Tables 5.2a – 5.2d.

Table 5.1: ECD table of the state vectors for the regexp "/(a|b)*cd/".

State	ECD 0	ECD 1	ECD 2	ECD 3
0	**012**	**012**	**012**	**012**
1	**12**	**-**	**-**	**-**
2	**-**	**3**	**-**	**-**
3	**-**	**-**	**4**	**-**
4	**-**	**-**	**-**	**-**

The process then generates 16 new ECD columns which are merged to form just 5 unique ECD columns of state vectors. The cross product computation is demonstrated in Tables 5.2a – 5.2d, and is as described in steps (vi) – (vii) of Algorithm 5.1. Also, The ECD columns of each of the tables in Tables 5.2a – 5.2d represent the state vectors described in Algorithm 5.1 step (vi). The cross product between the state vectors represented as ECD 0 and ECD 1 in step iv of Algorithm 5.1 ensures that all costly OR operations and consequently OR logic gates are eliminated. This is further reduces the logic circuits required during the hardware synthesis process.

Table 5.1 generates the ECD column labelled as ECD (0x1) seen in Table 5.2a. A similar process is executed to generate Table 5.2b – Table 5.2d. The fifth newly introduced ECD column was obtained as a result of the cross product computation, which generated a fifth ECD input. The input represents the new unique column of state vectors. Table 5.1 generated the 16 ECD columns of state vectors. Tables 5.2b – 5.2d show that the growth of the ECDs is linear and not exponential as is the case with most of the DFA and non-optimised NFA approaches discussed in Chapter 3.

After merging all the similar ECD columns and assigning to them unique 2-byte ECD class descriptors, a state vector for 2-byte ECDs was then obtained. Two 1-byte ECDs are then used to look up the table of 2-byte ECD as seen in Table 5.2e. The process is synthesised in the table-synthesis module of Figure 5.3, Figure 5.7 and Figure 5.9a.

The cross product between ECD 1 x ECD 2 = ECD (1x2) = 4 as seen in Table 5.2e, reflects the cross product between ECD (1x2) = 4 as seen in Table 5.2b. By recursively running the algorithm again as explained in step (viii) of Algorithm 5.1, and merging the various state vectors seen in Table 5.3a, the 4-byte ECDs are created. Two 2-byte ECDs are then used to look up the table of 4-byte ECD. We then end up with the Table 5.3b which is created from the cross product of the various 2-byte ECD columns of state vectors as seen in Table 5.2e. The table now has 6 ECDs represented by each the table columns. The algorithm that was used to create the various ECDs generated from the various class vectors is as described in Algorithm 5.1.

Table 5.2: (a) ECD 0 cross product of itself and those of ECD (1, 2 and 3).

State	ECD (0x0)	ECD (0x1)	ECD (0x2)	ECD (0x3)
0	**012**	**0123**	**012**	**012**
1	**12**	**3**	**-**	**-**
2	**-**	**-**	**-**	**-**
3	**-**	**-**	**-**	**-**
4	**-**	**-**	**-**	**-**

Table 5.2: (b) ECD 1 cross product between itself and those of ECD (0, 2 and 3).

State	ECD (1x0)	ECD (1x1)	ECD (1x2)	ECD (1x3)
0	**012**	**0123**	**012**	**012**
1	**-**	**-**	**-**	**-**
2	**-**	**-**	**4**	**-**
3	**-**	**-**	**-**	**-**
4	**-**	**-**	**-**	**-**

Table 5.2: (c) ECD 2 cross product between itself and those of ECD (0, 1 and 3).

State	ECD (2x0)	ECD (2x1)	ECD (2x2)	ECD (2x3)
0	**012**	**0123**	**012**	**012**
1	**-**	**-**	**-**	**-**
2	**-**	**-**	**-**	**-**
3	**-**	**-**	**-**	**-**
4	**-**	**-**	**-**	**-**

Table 5.2: (d) ECD 3 cross product between itself and those of ECD (0, 1 and 2).

State	ECD (3x0)	ECD (3x1)	ECD (3x2)	ECD (3x3)
0	**012**	**0123**	**012**	**012**
1	**-**	**-**	**-**	**-**
2	**-**	**-**	**-**	**-**
3	**-**	**-**	**-**	**-**
4	**-**	**-**	**-**	**-**

Table 5.2: (e) 2-byte table of compressed ECD input classes.

	ECD 0	ECD 1	ECD 2	ECD 3
ECD 0	**1**	**2**	**0**	**0**
ECD 1	**0**	**3**	**4**	**0**
ECD 2	**0**	**3**	**0**	**0**
ECD 3	**0**	**3**	**0**	**0**

Algorithm 5.1: Construction of n-ECD class vectors.

INPUT: An n-state, m-character class input. The input state is $s_.$

OUTPUT: An n-state ECD_RTS-NFA with the relevant multi-byte table of compressed ECDs.

BEGIN

i. Read and parse the regexps to be constructed into the equivalent ECD_RTS-NFA.

ii. For ∀ i < n, where i = 0,1,2,3....n-1, and n is the total number of states in the NFA. If the transition (link) from state s_i is a self-transition from state s_i to itself upon consuming a non-empty character, remove all such self-transitions.

iii. For ∀ i < n, j < n and k < n, if the output of state s_i connects to the state inputs of some state s_j upon consuming an empty string (ϵ), remove all such transitions $t_{i,j}$ linking state s_i to s_j. Create a new transition that connects state s_i to states s_j and s_k where $s_k > s_j$ on a non-empty input.

iv. For ∀ i < n, j < n and for each transition $t_{i,j}$ from a state s_i to a state s_j, scan through. Store all next states transited to on the same input, into a set of next states. Store all the different sets of next states into a single vector of sets of next states and assign a single input character class descriptor to them.

v. Assign to each classified inputs created in (iv) ECDs, which are the class descriptors. The ECDs now represent the sets of vectors of next states for all character classes that trigger transitions from a state s_i to a state $s_{j,}$ where i < n, j < n.

vi. Repeat steps (i) - (v) for ∀ s_i, i < n and store all the sets of vectors of next states in a list of state vectors for ∀ states s_i in the ECD_RTS-NFA.

vii. Once step (vi) is completed, the process of building the compressed table of ECDs begins. The process first performs the cross product computation of any two sets of vectors of next states v_i and v_j ∀ i < n, j < n contained within the list of state vectors stored in (vi). Subsequently, all the similar vectors are merged to become a single vector. Recursively performing step (vi) – (vii) generates a 4-byte table of ECDs two 2-byte tables.

viii. Finally, exit the process after step (vii) and generate the VHDL file for the ECD_RTS-NFA. The file is then uploaded to the XST VHDL synthesis tool for synthesis and implementation.

END.

During the second phase of the implementation phase described in Section 5.4.2a, two tables: Table 5.2e and 5.3b, representing 2-byte and 4-byte tables of ECDs respectively are then synthesised into logic in the table-synthesis module of Figures 5.3, 5.7 and 5.9a. The two tables are used to perform table look up operation (refer to Section 5.4.2a-ii for details).

Table 5.3: (a) Table of the 2-byte state vectors merged to form 6 new ECD columns.

State	ECD 0	ECD 1	ECD 2	ECD 3	ECD 4	ECD 5
0	**012**	**0123**	**012**	**0123**	**0124**	**0124**
1	**12**	**3**	**-**	**-**	**4**	**-**
2	**-**	**-**	**-**	**-**	**-**	**-**
3	**-**	**-**	**-**	**-**	**-**	**-**
4	**-**	**-**	**-**	**-**	**-**	**-**

Table 5.3: (b) 4-byte table of compressed ECDs.

	ECD 0	ECD 1	ECD 2	ECD 3	ECD 4
ECD 0	**0**	**1**	**2**	**3**	**4**
ECD 1	**0**	**3**	**2**	**3**	**4**
ECD 2	**2**	**3**	**2**	**3**	**5**
ECD 3	**2**	**3**	**2**	**3**	**5**
ECD 4	**2**	**3**	**2**	**3**	**5**

The second phase of the design implementation describes the basic hardware architecture of the ECD_RTS-NFA. Section 5.4.2 discusses the inner workings of the design and shows how each of the modules are integrated to form block of sub-REMEs.

5.4.2 Implementation of the Second Phase

This section is divided into five major sections namely: (1) hardware architecture module, 2) user constraint timings, (3) synthesis, and (4) target device configuration.

a. Hardware Architecture Module

i. Memory Modules

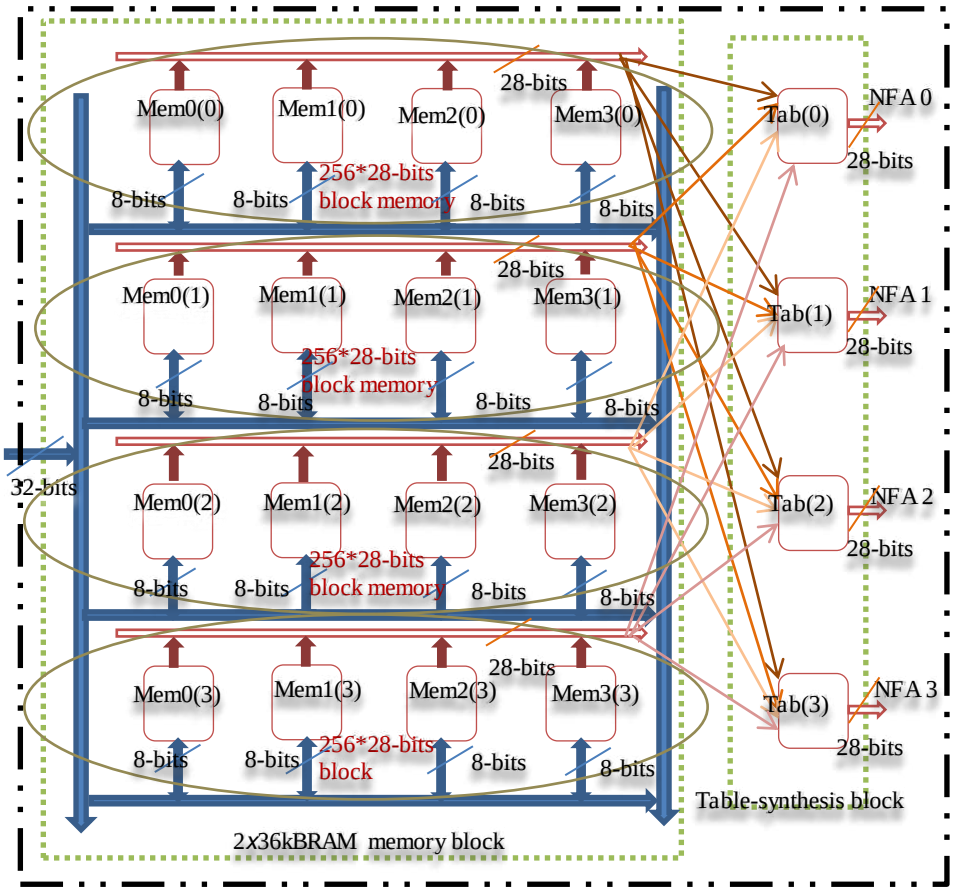
The BRAM referred to in this section is a primitive 36K BRAM block of memory. The BRAM block is constructed for compressing 1-byte ECDs. The ECDs are generated for an engine that matches between 8 - 20 regexps per REME. This is dependent on the complexity of the regexps in question as explained in Section 5.4.1. The 2x36kBRAM memory block interfaces with the table-synthesis and ECD_RTS-NFA block to form a single REME as shown in Figures 5.1 and 5.9a.

The memory design as shown in Figure 5.7 reduces the total memory overheads incurred by at least 90%, during the construction of each REME. The reduction is in comparison to the other related approaches. This was achieved by first splitting the two 36K BRAMs into 512x72-bits blocks. Each of the 512x72-bits is further split into two 256x72-bit blocks. This produced four 256x72-bit blocks, with each containing just 256x8-bits of ECDs. But because we are dealing with < 128 classes of ECDs (2^7 bits), the MSB (Most Significant Bits) of each 8-bit ECD input value is eliminated. This leaves us with 7 bits rather than 8 bits to represent each of the 4-byte ECDs. The 7-bit outputs from all the four blocks of 256x8-bit memories are then concatenated to form a 256x28-bit wide output of 4-byte ECDs. The ECDs are then supplied to the table-synthesis block module for processing as shown in Figure 5.7.

The REME block arrangement implies that, only two 36K BRAMs are required to represent 4-bytes of data inputs required for matching each of the four sub-REME NFA matching blocks. This translates to a significant reduction in memory in the optimised ECD_RTS-NFA design, compared to the initial ECD-NFA design. Initially, the ECD-NFA utilised a single 36KBRAM for just one regexp matching REME engine. The design required about twenty 36KBRAMs to effectively match a 20 regexp matching REME engine block, which was a waste of memory. Figure 5.7 shows the grid layout of the 2x36KBRAMs in more detail as illustrated in Figures 5.3 and 5.9a.

To properly comprehend the synthesis process for the memory modules described in Figure 5.7, one of the four 256x28-bit blocks of memory was extracted. An explanation is given on how the extracted memory block would be synthesised in a synchronised order, together with the other three remaining

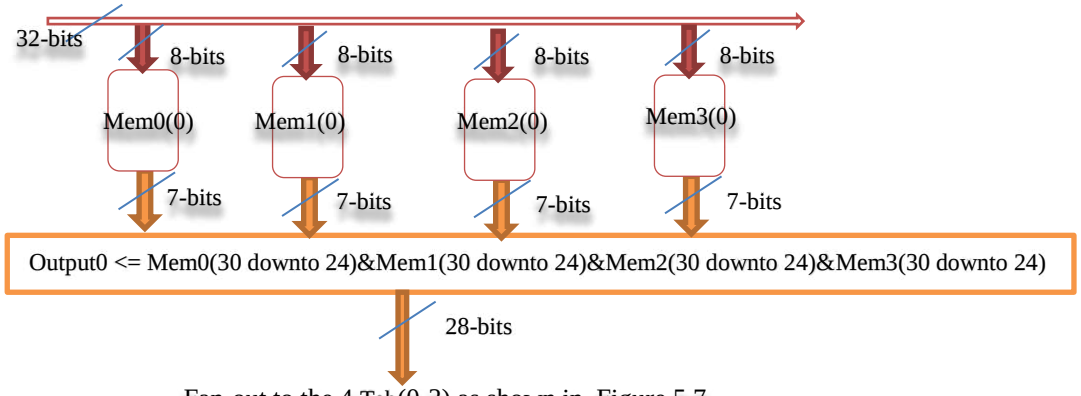
block memories. Figure 5.8a shows how the output of each Memx(0-3) tables was combined, where x = 0, 1, 2 and 3. The first of the 256x28-bit block memories namely: Memx(0) as shown in Figures 5.8a – 5.8d generated a 256x28-bit output belonging to Mem(0-3)(0) memory block. This was obtained by eliminating the MSB of each unit to produce a 7-bit output each. The four 7-bit outputs are then concatenated together to produce one 28-bit output.



One REME engine block.

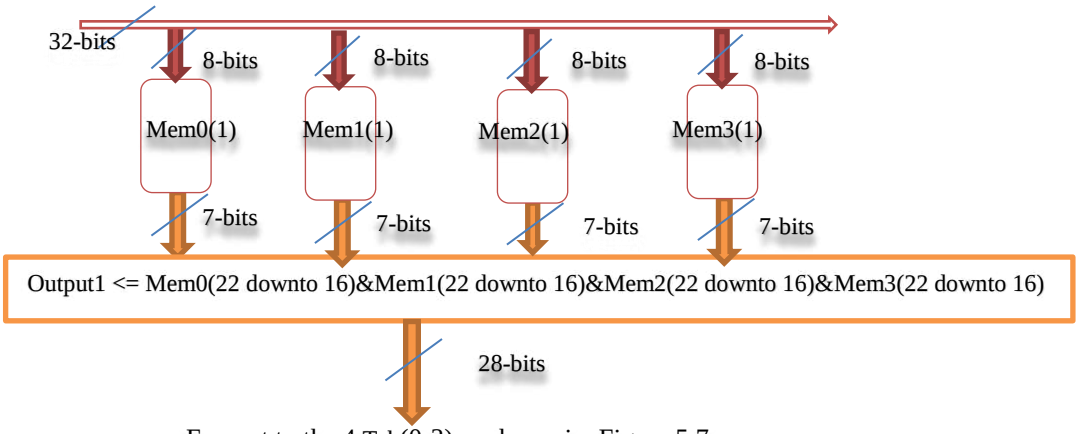
Figure 5.7: 2x36kBRAM block interface.

The process in Figure 5.8a is repeated for the other 3 memory tables: Mem1, Mem2 and Mem3 to also produce Output1, Output2 and Output3 all of which are 28-bits wide too. The outputs are also obtained by concatenating four of their separate 256x8-bit RAMs as shown in Figure 5.8b, Figure 5.8c and Figure 5.8d respectively:



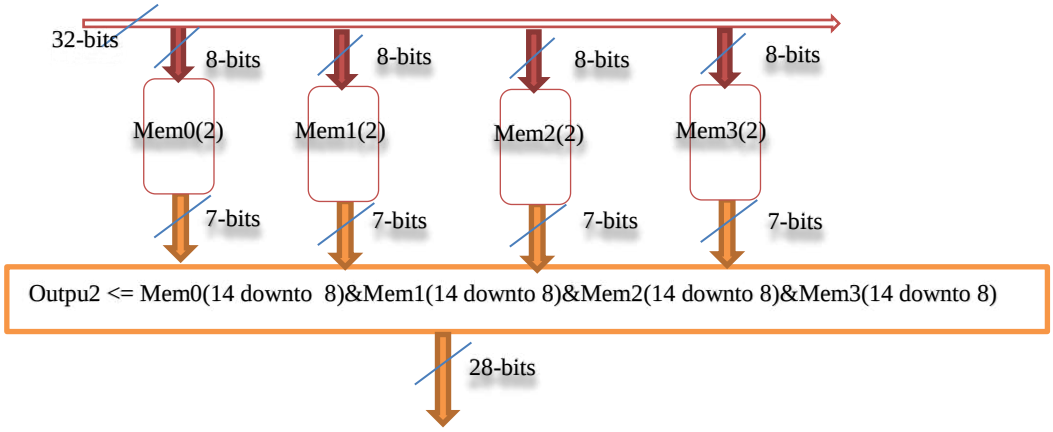
Fan-out to the 4 Tab(0-3) as shown in Figure 5.7.

Figure 5.8: (a) Four 256x8-bit table of ECDs for Mem(0-3)(0) memory blocks.



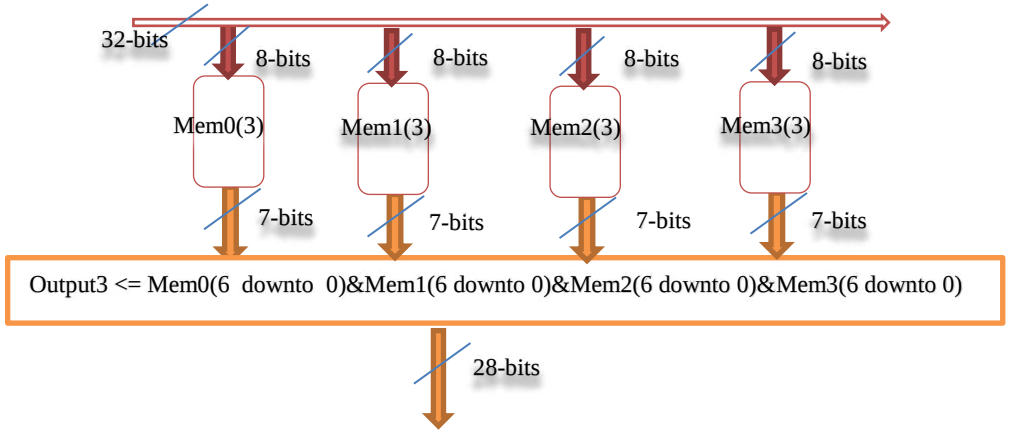
Fan-out to the 4 Tab(0-3) as shown in Figure 5.7.

Figure 5.8: (b) Four 256x8-bit table of ECDs for Mem(0-3)(1) memory blocks.



Fan-out to the 4 Tab(0-3) as shown in Figure 5.7.

Figure 5.8: (c) Four 256x8-bit table of ECDs for Mem(0-3)(2) memory blocks.



Fan-out to the 4 Tab(0-3) as shown in Figure 5.7.

Figure 5.8: (d) Four 256x8-bit table of ECDs for Mem(0-3)(3) memory blocks.

ii. Table-Synthesis Modules

The four table-synthesis modules as shown in Figure 5.9a have the same structure. Within the architecture body of each module resides the compressed multi-byte table of ECDs resulting from the cross product of any two ECD's state vectors. Each of the state vectors is then represented by a class descriptor which designates each unique input as an ECD, as discussed in Algorithm 5.1 of Section 5.4.1a. Further illustration on how the basic synthesis process works using a short VHDL code snippet contained Algorithm 5.2. The code snippet uses the 2-byte and 4-byte tables of compressed ECDs, generated for the regexp "/(a|b)*cd/" as presented in Section 5.4.1a. With the generated 2-byte tables and 4-byte tables of ECDs, the process of the synthesis starts.

The table-synthesis module as shown in Figure 5.9a contains the 2-byte and 4-byte tables of ECDs which are then synthesised into logic using LUT RAMs, multiplexers, shift registers and other related logic circuits. Executing the look up table process implies looking up 2 or 4 bytes of incoming 1-byte ECDs stored in the four sub-BRAM blocks. Afterwards, the module generates the vectors of < 128 bits representing each ECD that has been successfully scanned, for each of the four sub-blocks of the table-synthesis module. The < 128 bit vector outputs coming out of each sub-block are then passed on to each of the four sub-NFA blocks for matching. Each of the four sub-blocks in each category forms what is referred to as sub-REME blocks. Each sub-REME block is made up of a BRAM, table-synthesis module and a sub-NFA block which make up one REME block as shown in Figure 5.9a. If one of the Tab(i) sub-blocks of the table-synthesis blocks is expanded where i = 0, 1, 2, and 3, the hardware table-synthesis process can be executed as shown in Figure 5.9b.

From Figure 5.9b, $i_1 = j_1 = i_2 = j_2 = [0..6]$ bits, and each is an ECD input value. While k = Tab1(i_1, j_1) is a 7-bit value, and l = Tab1(i_2, j_2) is a 7-bit value. Tab1 is a table of 2-byte ECDs, while Tab2 is a table of 4-byte ECDs used to look up the ECD descriptors i_1, j_1, i_2, j_2. The inputs are fetched from the 2x36KBRAM memory blocks in Figure 5.9a recursively. Whenever a match is found, a bit value of 1 is assigned to the bit position of the output vector of < 128 ECDs representing the total number of ECD inputs in Tab2. The whole process is repeated until all the ECDs of Tab2 have all been scanned.

The algorithm for the table-synthesis process also describes Table 5.2e and 5.3b. Step iv of Algorithm 5.2 begins by first declaring and assigning signals i1, i2, i3, i4, and the input & output ports in the entity and architecture bodies of the VHDL code snippet. The step describes how the 2-byte and 4-byte table of compressed ECDs looks up multiple bytes from the stored 1-byte ECDs. The 1-byte ECDs are contained in the 2x36kBRAM block and are read into the table-synthesis block. Afterwards, the module generates the relevant < 128 bits of ECDs output in step v of Algorithm 5.2. The outputs are then fanned out to the various sub-NFA blocks as shown in Figure 5.8. The output port of Figure 5.9 is a standard logic vector. The size of the port is relative to the number of 4-byte ECDs generated for the sub-NFA blocks, during the parsing stage described in Algorithm 5.1.

The number of bit-vector positions in the output generated in Algorithm 5.2 has to be equal with the number of compressed ECDs expected by the sub-NFA blocks. The bit vector output of <128-bits is fanned-out to the four sub-NFA blocks for matching. Once again, the entire process involves a less costly table look up operation. Furthermore, due to the compact nature of the table-synthesis approach, the increased complexity of regexps utilised only has a linear effect on the overall growth of the design REMEs.

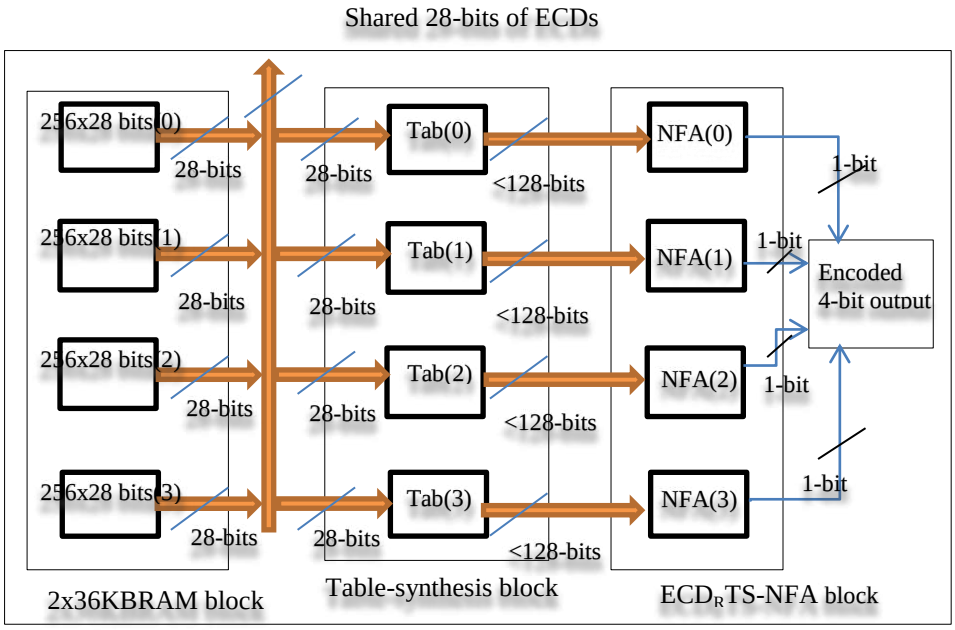


Figure 5.9: (a) An expanded REME diagram as shown in Figure 5.7.

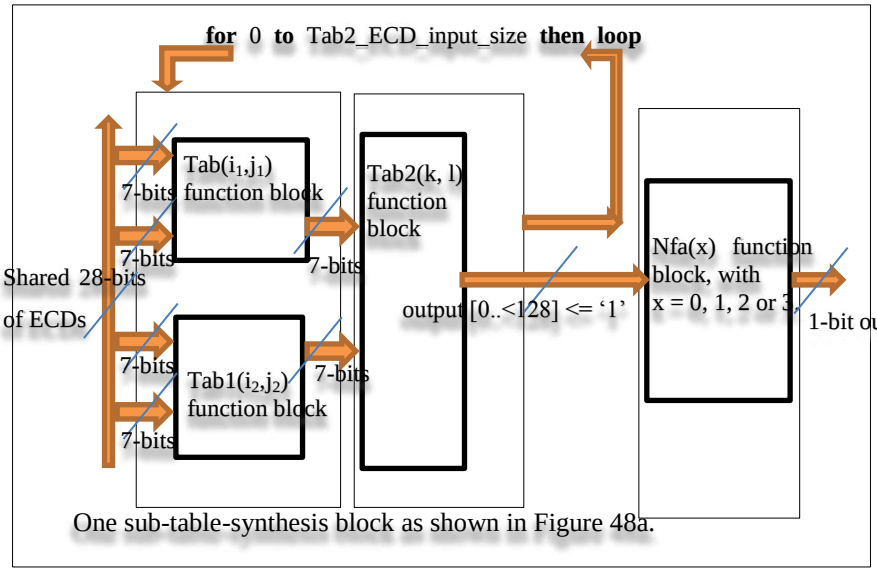


Figure 5.9: (b) One of the sub-NFA blocks of Figure 5.9a.

Chapter 6 discusses the various results obtained for each category of the n-regexp REME designs, where n = 2, 3, 4 and 5. The design does not scale beyond 5-regexp REMEs at the moment.

Algorithm 5.2: Hardware synthesis process for the compressed n-byte ECDs.

INPUT: An k *x* k table of n-byte ECD input class descriptors and a 28-bit input from the 2x36kBRAM block of Figure 5.9a, where n = 2 and 4, and k > 1.

OUTPUT: A <128-bit vector of compressed ECDs.

BEGIN

i. Read the 28-bit inputs from the 2x36kBRAM and the k *x* k tables of n-byte ECDs.
ii. Create the relevant 2-dimensional arrays converted into signal variables and initialise the same to contain the associated 2-byte and 4-byte tables of compressed ECDs.
iii. Compute and process the sub-linear table-look up operations, to generate the relevant < 128-bit vector of outputs. Each bit position of the output bit vector represents an equivalent ECD value.
iv. Initialise the tables of 2-byte and 4-byte tables of compressed ECDs. Assign the 1-bit value of '1' to the output variable.

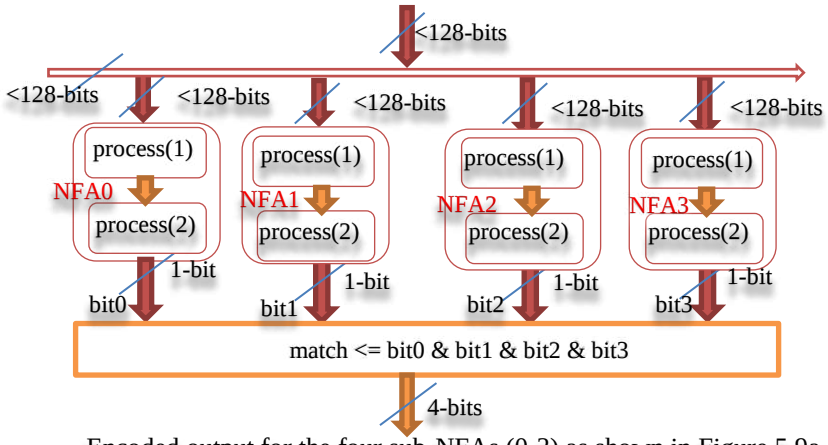
END.

iii. ECD_RTS-NFA Module Blocks

From the ECD_RTS-NFA block in Figure 5.9a, it can be seen that there are four sub-NFA blocks. Each of the four separate blocks represents a sub-REME that matches up to 5 regexps depending on the complexity of the regexps and design considered. The regexps used for building and testing these ECD_RTS-NFA sub-blocks were selected from the VRT Rule distributed by Sourcefire (v 2.0) community rules, 2001-2013 (Snort 2013; Sourcefire 2009) provided by Snort. The rules selected contain attacks such as: malware-backdoor PROTOCOL-FTP CWD ~root attempt, PROTOCOL-FTP no password/bad login, POLICY-OTHER FTP anonymous login attempt, SERVER-MAIL RCPT TO overflow, FILE-IDENTIFY.htr access file download request and many more.

Each of the four sub-NFA modules has the same structure and only varies in the number of states and ECDs generated. Algorithm 5.3 shows how the ECD_RTS-NFA sub-blocks work. Figure 5.10 illustrates the way the four sub-blocks are closely coupled together based on the overall ECD_RTS-NFA block, with each containing the four sub-NFA as shown in Figure 5.9a. The two-phased process as shown in Figure 5.10 is computed in step iii of Algorithm 5.3. The step first describes the process responsible for initialising the state signal variables and processing the ECDs. The process drives the automation's transitions from one or multiple current states to one or multiple next states. This is achieved by consuming a single or multiple ECDs capable of activating multiple states at once, as expected of a typical NFA operation.

Also, step v of Algorithm 5.3 is responsible for generating the < 128 bits of ECDs from the table-synthesis module as shown in Figure 5.10. The ECDs are then fanned-out to the various sub-NFA blocks. The port ip is a bit vector of < 128 bits of ECDs that are supplied the sub-NFA blocks as inputs. The CSV represents the current state bit-vector, while the NSV represents bit-vector of next states. Lastly, Figure 5.10 illustrates how each of four sub-NFA blocks of the ECD_RTS-NFA is further laid out as shown in Figure 5.9a. Each of the ECD_RTS-NFA sub-NFA blocks as shown in Figure 5.10 is made up of the two-phased process described in Algorithm 5.3. The four separate 1-bit outputs are then concatenated together to produce a 4-bit match vector for every single REME engine block as shown in Figure 5.9a. By recursively repeating the processes described in Algorithm 5.1, 5.2 and 5.3 all the relevant REMEs for each of the REME designs then generated.



Encoded output for the four sub-NFAs (0-3) as shown in Figure 5.9a

Figure 5.10: Two-phased process for each sub-NFA block.

The algorithm used to describe the process of creating the creating the ECDRTS-NFA blocks to perform the matching process is described in Algorithm 5.3.

Algorithm 5.3: Hardware synthesis process for the ECDRTS-NFA block.

INPUT: A <128-bit of n-byte ECDs supplied to the NFA from the table-synthesis block of Figure 5.9a, where n = 2 and 4.

OUTPUT: A k-bit vector of match output, where k < 128.

BEGIN

i. Read the <128-bit vector of n-byte ECDs from the table-synthesis block of Figure 5.9a.
ii. Create the necessary signal variables and initialise them as: current states, next states and the variable that holds the total number of states on each sub-NFA.
iii. Compute the two-phased processes as shown in Figure 5.10.
iv. Generate the associated 1-bit match output against each of the four sub-NFA blocks of the overall ECD_RTS-NFA block as shown in Figure 5.9a. Finally, encode all the outputs as a 4-bit vector of outputs.
v. Generate the < 128 bits of ECDs from the table-synthesis module as shown in Figure 5.10

This arrangement reflects the one portrayed in Figures 5.9a, 5.9b and 49.

END.

iv. Top Module

The top module has an entity, architecture and processes code section as explained in Section 2.5. The top module instantiates and constructs all the various implementations described in Section 5.4.1. A separate process FF is used to speed up the transfer of the output of each table-synthesis sub-blocks, to the corresponding sub-NFA blocks as shown in Figure 5.9a. Afterwards, a match vector is generated by concatenating the various match outputs of the overall sub-NFAs and encoded accordingly. The test bench module specifies in VHDL the role of a complete simulation environment for the analysed system UUT (refer to Section 5.3.2a). It contains both the UUT as well as the stimuli for the simulation. The module also has a code section for the implementation of the top module.

In the body of the entity a generic value for the word size of 4-bytes based on the architecture was initialised. An array of network data, representing the decimal equivalent of the ASCII character codes, was set and used to provide the stimuli for the top level module. The UUT module was then instantiated,

and afterwards a generic clock signal was set through a simple process. Another process that copies bytes of the network data into a single standard logic vector was then initiated. The final process involves fetching 4-byte data, and passing it to the output port of the instantiated top level module for onward passage to the BRAM block module as illustrated in Figure 5.7.

b. User Constraint Timings

This process involves a pre-synthesis and post-synthesis procedure, where the timing specification is set by the user or by default. There is a user constraint file (UCF) editor which allows the user to set the clock timing. The NET connectivity identifies groups of elements by specifying a net or signal that drive synchronous elements and pads. Such synchronous elements include: FFs, Latches, BRAMs, distributed RAMs etc. CLK is a short name for a CLOCK net. The TNM_NET is the equivalent timing name (TNM) on a net constraint, with the exception of the input pad nets. The timing name is used to identify the elements of a group used in a timing specification as described by Xilinx (2012b, p. 462).

The PERIOD constraint is a basic timing and synthesis constraint used to check timings between all synchronous elements within a clock domain. The domain is usually defined in the destination element group (Xilinx 2012b). The HIGH (rising clock edge)|LOW (falling clock edge) keyword of the PERIOD constraint defines the initial clock edge (RISING|FALLING) for analysis of OFFSET constraints. The definition The HIGH|LOW value is set to 50% duty cycle, which is the default value for most clocks.

A user constraint file (UCF) is generated and added to list of files attached to the top level module in preparation for synthesis. When the option for choosing the option to ignore the user timing constraints is selected in the properties environment, the system automatically synthesises the design. This is achieved by determining the best applicable and estimated timing constraint for the design's clock. The default timing is not exactly accurate when set automatically in comparison to the user specific timing constraints. The clock value for CLK had to be adjusted each time the design is synthesised to obtain the minimal timing requirement for each design. The two-line statements that describes how the timing editor outputs the clock timing based on the user settings is stated thus:

```
NET "CLK" TNM_NET = CLK;
TIMESPEC TS_CLK = PERIOD "CLK" 5.778 ns HIGH 50%;
```

The initial timing settings contained within the UCF is initialised and subsequently adjusted to achieve the best user-defined timing constrain.

c. Synthesis

For a full description of the synthesis process, refer to Section 2.7. Before the process of synthesis is initiated, a few settings were set using the process property for synthesis options. The three affected categories are: the synthesis options, the HDL options, and Xilinx specific options. The Synthesis process is somewhat systematic and begins by firstly having the VHDL codes synthesised or translated into a netlist (refer to Section 2.5). This is achieved using the installed XST VHDL synthesis tool software bundled within the family of Xilinx FPGA Virtex-6 device (refer to Section 2.7 for details).

In summary, the XST VHDL synthesis tool simply performed the RTL parsing (refer to Section 2.5) and circuit elaboration of the design. The tool also carried out some register/logic level optimisations (Xilinx 2012b, p. 23; Jang et al. 2009) and performed behavioural and post (translate, map, place and

route) simulations. The generated netlist file is saved as an NGC file. The NGC file served as input to the place and route process described in Section 2.7.

d. Configuration File Generation

The synthesised netlist (NGC file) generated in Section 5.4.2c is then implemented. The NGC file is converted into a binary format, where the components and connections that it defines are mapped to CLBs. The translation phase of the implementation stage generated a post-translate simulation model. At the translation phase, a process merged all of the input netlists and design constraints files for the design. The process finally generates an NCD output file. The NCD file was used to physically represent and map the design to the Xilinx FPGA logic components. Closely following the mapping phase was the place and route (PAR) phase. The phase takes the mapped NCD files, and then places and routes the design (refer to Section 2.7 for more details).The final phase in the implementation stage is the programming file generation phase. The phase involves the process of generating the configuration bits file (Xilinx 2010, pp. 136-137) used by the target Xilinx FPGA Virtex-6 device.

5.5 Chapter Summary

In this chapter a novel equivalence classification based approach to regexp pattern matching was presented. The approach is simple and less complex, and is capable of performing basic optimisations capable of creating classified inputs using a compressed and minimise NFA-based REMEs. The fact that the design is modular in approach and capable of utilising a single bit output to represent a class of multiple possible matching characters is incredible. The combined optimisation techniques used helped create a design that is uniquely applicable to NFAs only. The approach is capable of reducing the overall table size of the ECDs used to drive the design REMEs. The memory blocks also utilised within each REME only 2x36KBRAMs for up to 5-regexp sub-REMEs, instead of 20x36KBRAMs for every single regexp sub-REME. This reduction represents a 90% reduction in the total number of BRAMs that are required per each sub-REME. Building such efficient REMEs remains a challenge with most NFA-based hardware approaches. Chapter 6 discusses how the results of the design in this thesis vary across the four separate REME designs and how it compares favourably with the other related approaches studied in this thesis. The test of the research hypothesis is also discussed in Chapter 6.

6. Evaluation of Results

This chapter examines the various results obtained from the design and implementation process described in Chapter 5. Afterwards, the results of the related approaches explained in Chapter 4 are tabulated, compared and analysed together with the design results obtained in this thesis. This helps to establish the performance of the new approach. The discussion concludes with a brief summary of the factors that may be considered as design limitations.

6.1 Design Description

This chapter describes the results and the processes used to implement the various REME designs in this thesis, with particular reference to Section 5.4 of Chapter 5. You may recall that each REME block contains four separate sub-REME blocks. The various results for the four REME designs are as shown in Appendix 1.3 – 1.6. Also, the description of the number of ECDs and the total number of characters matched is as shown in Table 6.1 and Table 4.6 of Chapter 4. The four REME designs show how the approach in this thesis scales up with each increase in the number of regexps that are built into the ECD_RTS-NFAs design thus:

1. BG2RE (denotes a 4-byte ECD_RTS-NFA REME that matches up to 2 regexps).
2. BG3RE (denotes a 4-byte ECD_RTS-NFA REME that matches up to 3 regexps).
3. BG4RE (denotes a 4-byte ECD_RTS-NFA REME that matches up to 4 regexps).
4. BG5RE (denotes a 4-byte ECD_RTS-NFA REME that matches up to 5 regexps).

6.2 Design Results

The various results of the ECD_RTS-NFA designs are presented in the following section. This helps to portray the effect of the optimisations performed on the original ECD-NFA design. The section also shows how the various FPGA-based related approaches compare with the ECD_RTS-NFA.

6.2.1 ECD_RTS-NFA and Related Approaches Result

In this section, each of the results presented below is considered based on the same attributes discussed in Chapter 4 namely: the design approach and an n-byte match size, with n = 1, 2 and 4, with focus on 4-byte matching designs considered to be the standard for comparison in this area of research. Also considered are the speed of matching (MHz), throughput (Gbps), and the total number of characters matched. The results appear in tables and are analysed by means of simple diagrams. Also the outcome of the research hypothesis in this thesis is discussed at the end of the chapter. The column headings of Table

6.1 have the same description and meaning as those of Table 4.1 - 4.6 of Chapter 4. Each column represents the important fields considered later on in Section 6.2.2b as shown in Table 6.1.

In Table 6.1 the last four entries show the 4-byte ECD_RTS-NFA REMEs namely: **BG2RE, BG3RE, BG4RE,** and **BG5RE**. The acronym 'BGnRE' simply means 'block-engine group of n regexps, where n = 2, 3, 4 and 5'. Four separate blocks of sub-REMEs make up each REME matching block. Each BGnRE contains 10 REME engines arranged in parallel to perform pattern matching as shown in Figure 5.9a of Chapter 5. Furthermore, the BGnRE design matches up to 5 regexps at a time, with each REME made up of 4 x n regexps sub-REME. Table 6.2 shows the results of processing the set of n regexps for each sub-REME, and the total number of ECDs computed in each category. The table also shows the average number of ECDs per sub-REME reported for each BGnRE column as seen in Table 6.1.

Table 6.1: The design approach compared with other related approaches as seen in Table 4.6.

Design Approach	Input (bytes)	MHz	Tp	T/Chars
Brodie, Taylor and Cytron (2006).	4	133.00	4.26	11126
Sourdis and Pnevmatikatos (2004).	4	303.00	9.71	18032
Yamagaki, Sidhu and Kamiya (2008).	4	113.40	3.63	40896
Hutchings, Franklin and Carver (2002).	1	30.90	0.24	8003
Lee, Hwang, and Park (2007).	2	275.30	4.40	19275
Lin et al. (2006).	1	133.00	1.10	20914
Hieu et al. (2011).	1	231.25	1.85	13287
Clark C.R and Schimmel E. D (2003).	1	253.00	2.00	17537
Sutton (2004).	4	317.19	10.15	2016
Clark and Schimmel (2004).	4	218.90	7.00	17537
Mitra, Najjar and Bhuyan (2007).	16	100.78	0.81	10977
Yang, Jiang and Prasanna (2008).	4	233.13	7.46	15000
Yang and Prasanna, (2009)	4	300.00	9.60	28000
Yang and Prasanna (2012).	4	198.6	6.36	120000
Yang and Prasanna (2012).	4	166.7	5.33	100000
Ganegedara, Yang and Prasanna (2010).	4	202.90	6.50	16384
Long et al. (2011).	1	155.50	1.24	1020
Singapura et al. (2015).	8	340.63	21.8	100000
BG2RE.	4	367.34	11.44	8274
BG3RE.	4	308.04	9.63	10229
BG4RE.	4	293.71	9.18	10287
BG5RE.	4	264.32	8.26	11127

6.2.2 Data Analysis of All Designs

The analysis is divided into two sections. Section 6.2.2a represents the results as seen in Table 6.1, while and 6.2.2b represents the results of Table 6.2 and Table 6.3 combined. The analysis of the results also includes the attribute referred to as: **Throughput efficiency** (**Tpe**), which is computed and discussed in

Section 6.2.3**.** Table 6.2 shows the number of ECDs generated by each 'BGnRE', while Table 6.3 shows the total number of characters matched by each 'BGnRE'. Both Table 6.2 and Table 6.3 were generated during implementation in the XST VHDL synthesis tool, and the target device is the Xilinx FPGA Virtex-6 device, which are all bundled in the Xilinx ISE FPGA Project Navigator, version 14.4 design suites.

a. Analysis of the Combined Designs

The basic concern in this thesis is to improve the throughput of the design. This is achieved while attempting to reduce the high logic circuit cost as obtainable in the other related approaches as reported in Chapter 4. Reducing processing time and storage costs is necessary in order to obtain good throughput efficiency. This is a serious challenge that all the related approaches are attempting to address. The diagrams reported in this section are based on the results reported in Table 6.1. Figures 6.1 - 6.4 show the various graphs reported against the various designs as they appear in Table 6.1.

Figure 6.1 shows that about 73% of the designs were not able to get beyond the average speed range of 270MHz – 280MHz in their respective designs. This is because a design with a speed beyond such a range reflects good performance. The least recorded speed obtained and attributed to the poorest BG5RE design is 264.32MHz. The highest speed of 367.34MHz was recorded against the BG2RE design, which happens to be the best. In Figure 6.1, just about 27% of the designs were able to cross the 300MHz mark. It shows that the BG2RE design with a speed of 367.34MHz is about 14% faster than the speed of 317.19MHz realised by the Singapura et al. (2015). The BG2RE is also about 92% faster than speed of 30.90MHz reported by Hutchings, Franklin and Carver (2002)**.**

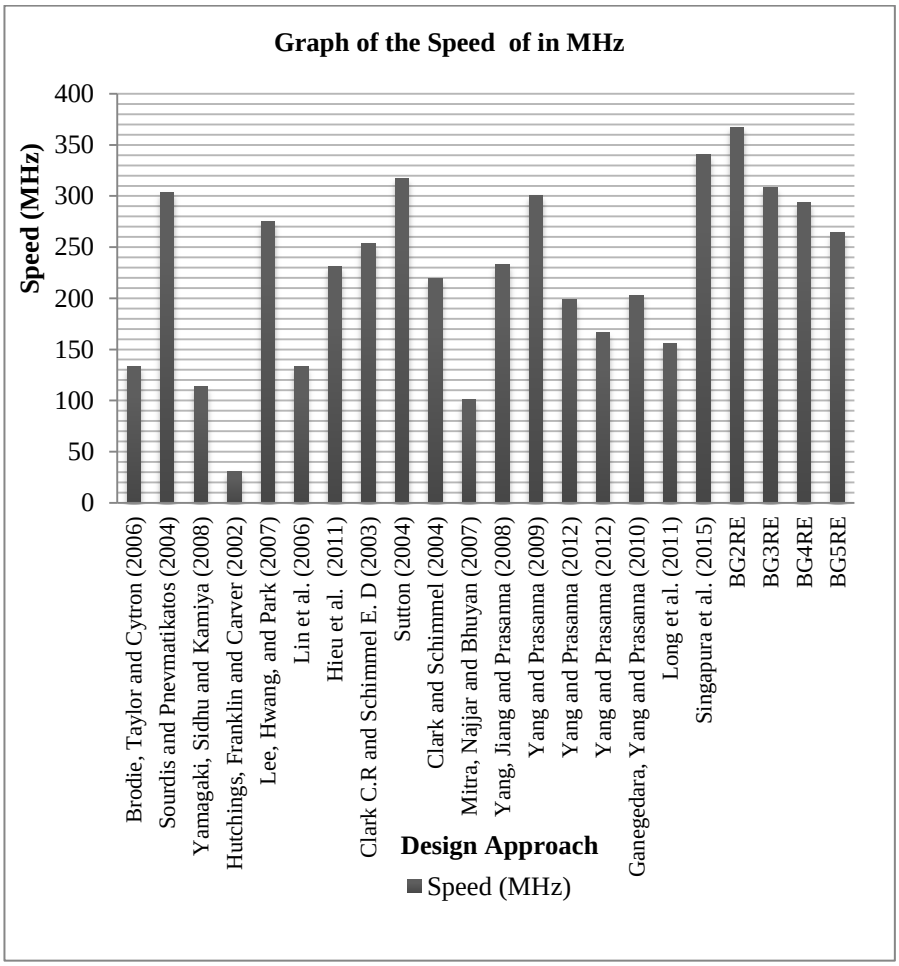


Figure 6.1: Graph of the speed (ranging from 30.90MHz - 367.34MHz).

Figure 6.2 shows that the average throughput obtained by about 41% of the designs is between the ranges of 4.00 Gbps - 4.50 Gbps. However, 41% of them have a throughput value between the ranges of 7.5 Gbps – 10 Gbps. Those designs with 4 Gbps throughput clocked at speeds below 150MHz, while those between the ranges of 7 Gbps – 10 Gbps achieved speeds between the 200MHz – 340MHz mark. The BG2RE achieved a throughput of 11.44 Gbps, which was the fastest compared to any of the other 4-byte REME designs. The least throughput reported among the designs was that of the BG5RE design. The BG5RE has a throughput of 8.26 Gbps, which is still better than about 63% of the other approaches as shown in Figure 6.2. However, the design by Singapura et al. (2015) is an 8-byte REME design, which explains the high throughput value of 21.8 Gbps that was reported as shown in Figure 6.2

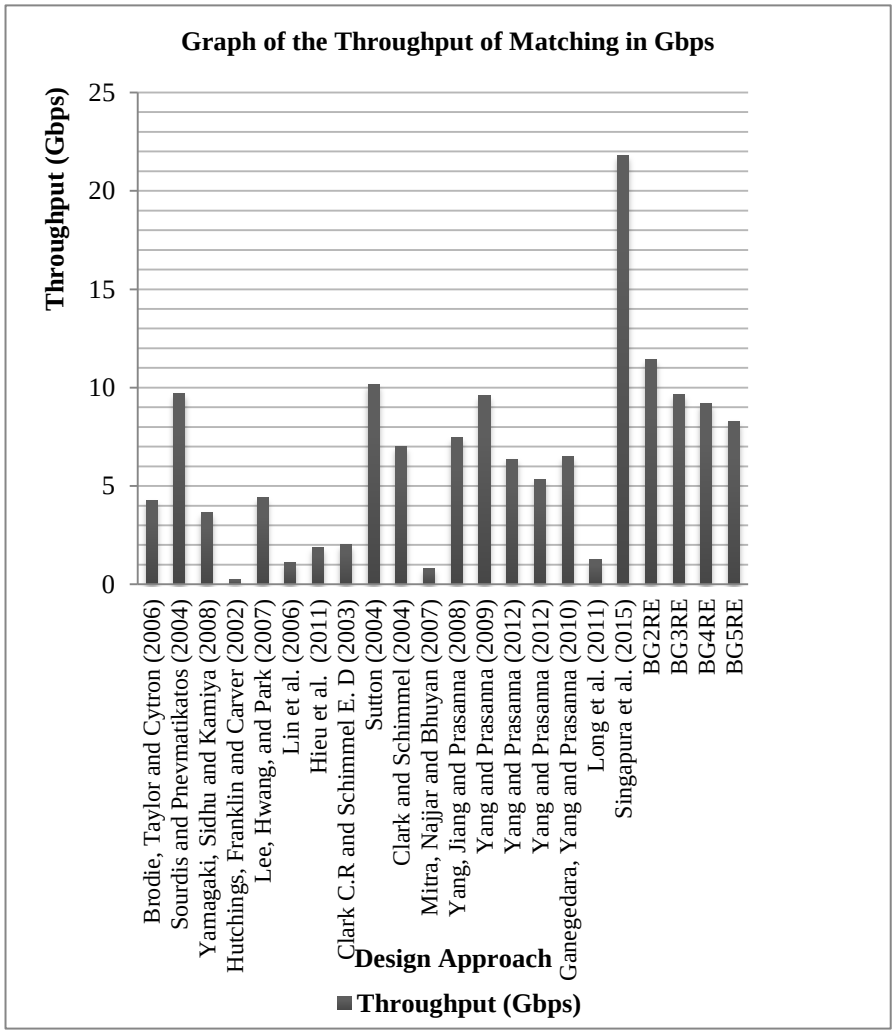


Figure 6.2: Graph of the throughput (ranging from 0.24-11.44 Gbps).

Figure 6.3 shows that the largest concentration of the total number of characters matched by about 77% of the designs is between the ranges of 10,000 – 20,000. The number represents characters matched by even the most efficient and best performing 4-byte REME designs. It also implies that matching characters beyond that threshold does not necessarily translate to better performance, when compared against the speed and throughput generated by the various designs.

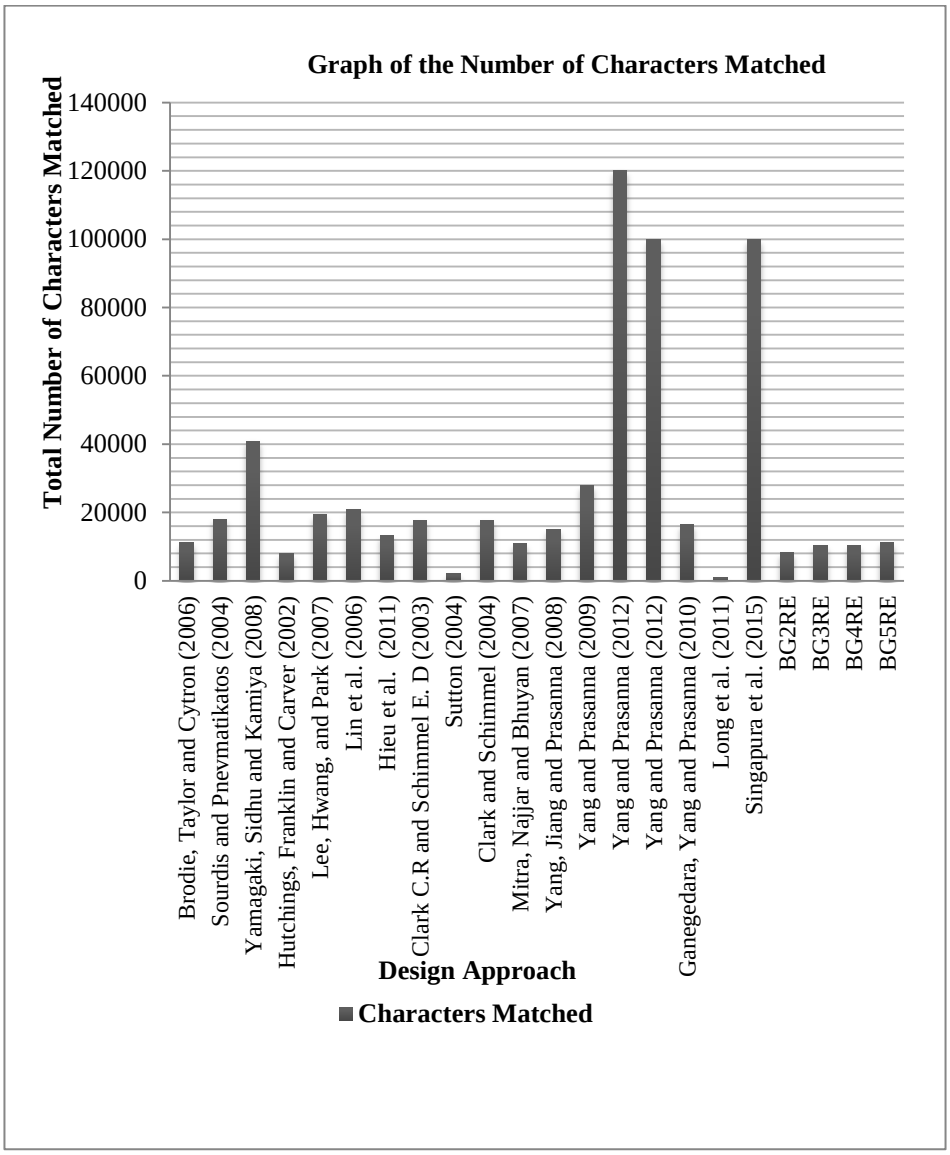


Figure 6.3: Graph of the total number of characters matched (ranging from 652-120,000).

Figure 6.4 shows that about 77% of the designs implemented a multi-character matching design, and about 63.7% of them implemented a 4-byte wide character matching scheme which is considered to be the standard used. The aim of each design is to improve its overall throughput. Mostly, the throughput is obtained at the expense of higher logic circuit cost. Furthermore, a 4-byte matching design was mostly utilised by the various designs in order to balance resource utilisation with throughput of matching. Mitra, Najjar and Bhuyan (2007) implemented a design having sixteen 1-byte matching REMEs. The design is made up of a 16-byte wide matching unit, but because the overall throughput is split amongst the units, the design REME throughput is only 0.81 Gbps. Also, Singapura et al. (2015) reported an 8-byte matching REME design, with an overall throughput of 21.8 Gbps.

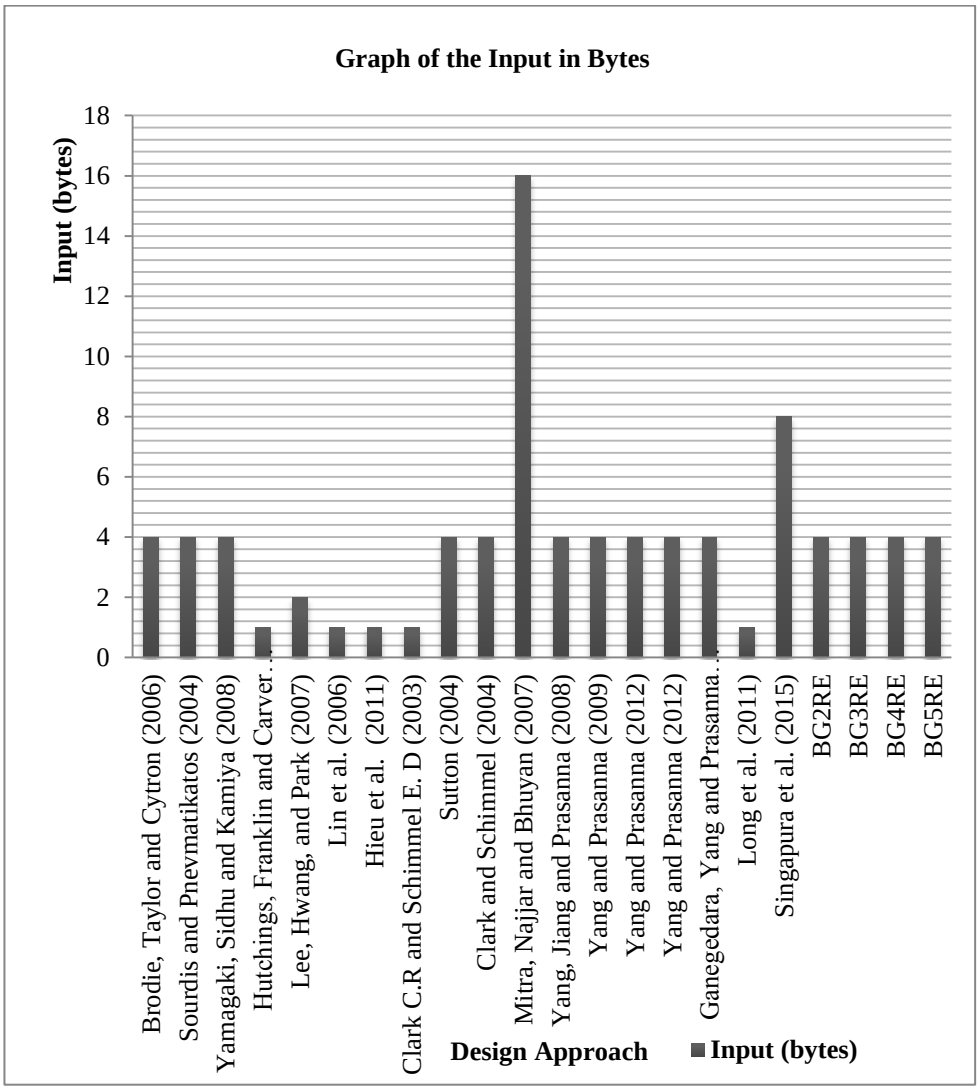


Figure 6.4: Distribution of the Input (n-bytes, n = 1, 2, 4, and 16).

b. Evaluation of the ECD_RTS-NFA REMEs

This section is based on the results reported in Table 6.2 and Table 6.3. It establishes the relationship between the combined ECDs and the number of characters utilised per REME block. The relationship is used to establish some pattern, as the design is scaled up from BG2RE to BG5RE engines. The short form of 'BGnRE Average ECDs' simply means 'the average number of ECDs for each BGnRE engine', as shown in Table 6.2. The short form 'BGnRE 4-byte CH' simply means 'the sum of the characters matched per clock cycle by each 4-byte BGnRE engine, with n = 2, 3, 4, and 5 as shown in Table 6.3.

i. Graphs of the Distribution of ECDs

Figure 6.5 - Figure 6.8 shows the distribution of the various ECDs reported for each 4-byte BGnRE engine as they appear in Table 6.2 and Table 6.3. The four categories of REMEs have 10 sub-REMEs each. The plots also show the distribution of the various ECDs obtained for the different BGnRE engines. Also, each of the sub-REMEs contains four parallel matching units arranged in parallel as explained in Section 5.3.

Table 6.2: Table of 4 x n-regexp ECDs and their averages.

TYPE	2RE ECDs	AVG	3RE ECDs	AVG	4RE ECDs	AVG	5RE ECDs	AVG
ENG1	20		56		53		70	
	34		28		69		61	
	22	**23**	24	**32**	53	**56**	65	**60**
	14		19		48		41	
ENG2	16		44		103		60	
	20		36		63		82	
	37	**29**	22	**32**	59	**66**	73	**64**
	40		26		39		41	
ENG3	42		47		39		52	
	8		35		45		87	
	39	**27**	33	**39**	40	**45**	58	**61**
	18		41		54		47	
ENG4	32		41		53		66	
	32		49		55		68	
	24	**29**	38	**39**	71	**60**	68	**74**
	25		28		58		94	
ENG5	24		61		54		75	
	63		36		75		79	
	34	**37**	45	**50**	79	**65**	92	**76**
	24		55		52		55	
ENG6	33	**22**	24		48		70	
	15		48		74		95	
	18		26	**36**	47	**53**	65	**80**
	19		43		40		90	
ENG7	11		40		37		58	
	37		45		58		75	
	28	**29**	43	**44**	66	**51**	63	**74**
	39		48		40		99	
ENG8	28		25		37		83	
	8		49		44		44	
	29	**23**	35	**46**	76	**49**	60	**63**
	25		74		37		65	
ENG9	24		57		67		58	
	30		42		37		76	
	21	**28**	48	**49**	62	**56**	86	**72**
	35		47		57		65	
ENG10	23		33		63		92	
	23		29		66		78	
	37	**28**	33	**36**	36	**58**	55	**83**
	28		48		65		107	

Table 6.3: 4 x n-regexp total number of characters matched with their sums per engine.

Type	2RE Chars	Sum	3RE Chars	Sum	4RE Chars	Sum	5RE Chars	Sum
ENG1	15	**925**	362	**752**	295	**1151**	330	**1323**
	630		26		344		473	
	20		93		344		76	
	260		271		178		444	
ENG2	22	**1300**	37	**605**	474	**1811**	60	**1151**
	697		117		688		320	
	453		23		597		476	
	128		428		52		295	
ENG3	228	**319**	394	**886**	38	**802**	254	**1736**
	37		125		235		380	
	37		317		301		322	
	17		50		228		780	
ENG4	320	**875**	50	**230**	39	**703**	73	**485**
	49		40		47		98	
	478		79		558		182	
	28		61		59		132	
ENG5	24	**526**	145	**411**	228	**536**	203	**925**
	142		46		59		52	
	330		45		54		156	
	30		175		195		514	
ENG6	521	**782**	52	**2761**	315	**765**	309	**1492**
	13		286		364		488	
	21		98		43		368	
	227		2325		43		327	
ENG7	12	**692**	1048	**1320**	342	**876**	333	**1465**
	284		45		353		676	
	275		44		138		361	
	121		183		43		95	
ENG8	472	**1266**	220	**481**	342	**1487**	160	**882**
	15		44		892		55	
	331		163		193		314	
	448		54		60		353	
ENG9	64	**337**	964	**1198**	59	**1114**	212	**648**
	26		51		630		194	
	23		135		286		173	
	224		48		139		69	
ENG10	314	**1252**	35	**800**	397	**1038**	94	**1020**
	633		324		359		514	
	281		258		136		232	
	24		183		146		180	

Figure 6.5 shows the plot for BG2RE ECDs against each of the 10 REME engines. The figure also shows that 60% of the ECDs fall within a controlled range of 27 - 28 ECDs. Figure 6.6 represents the graph for the BG3RE ECDs. The figure was plotted against each of the 10 REME engines. Figure 6.6 shows that, about 60% of the ECDs fall within the controlled range of 38 - 52. This shows a steady rise in the number of ECDs as expected, compared to ones in Figure 6.5. This is attributed to the increase in the number of regexps contained in the BG3RE engines. Each sub-REME contained in the 10 BG3RE engines is implemented with 3-regexps each. This is a deviation from the 2-regexps implemented in each of the sub-

REMEs contained within the BG2RE engines. The same applies to the BG4RE and the BG5RE engines with each implementing 4 and 5 regexps respectively.

For the BG4RE engines about 80% of the number of ECDs steadily rose and fell mostly within the controlled range of 50 - 60 as shown in Figure 6.7. This also shows a steady rise in the number of ECDs compared to the ones reported in Figure 6.6. About 60% of the number of BG5RE ECDs fell within the controlled range of 60 - 83 for the BG5RE engines as shown in Figure 6.8. This steady and not drastic rise in the number of ECDs shows efficiency of the overall ECD_RTS-NFA REME approach.

It is expected that an increase in the number of regexps contained in each REME category will lead to an abnormal growth in the number of ECDs generated. This is a common problem with most traditional DFA and non-classification based NFA approaches. Complex regexps consisting of multiple wildcards with length restrictions beyond a 1000 are easily avoided by a lot of approaches. However, the ECDRTS-NFA REMEs implemented in this thesis have effectively contained the abnormal growth of the NFA states, transitions and the ECDs generated in the overall approach as shown in Figure 6.5 - Figure 6.8.

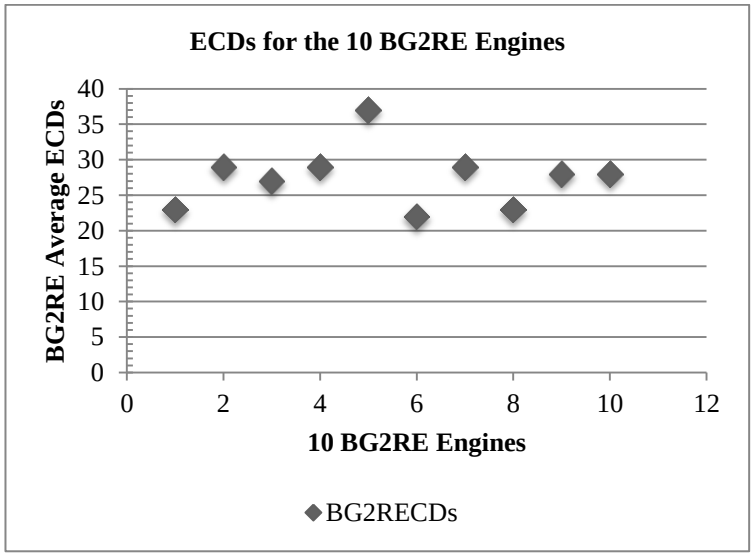


Figure 6.5: Distribution of BG2RE average ECDs for the 10 BG2RE engines.

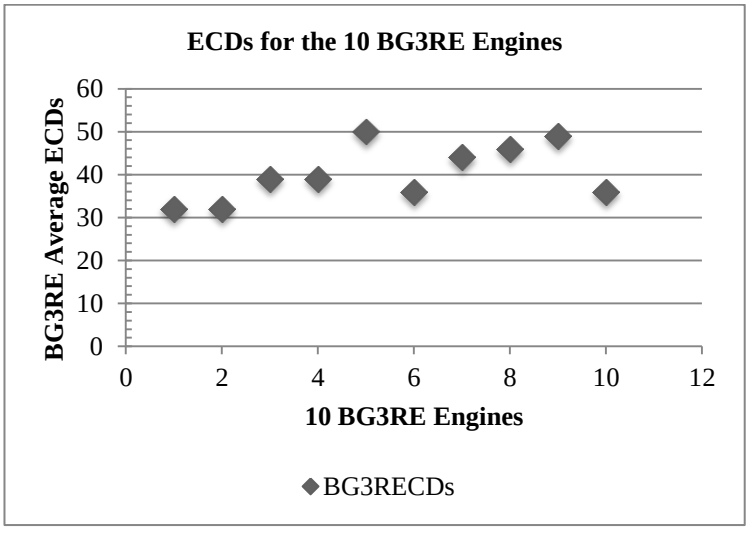


Figure 6.6: Distribution of BG3RE average ECDs for the 10 BG3RE engines.

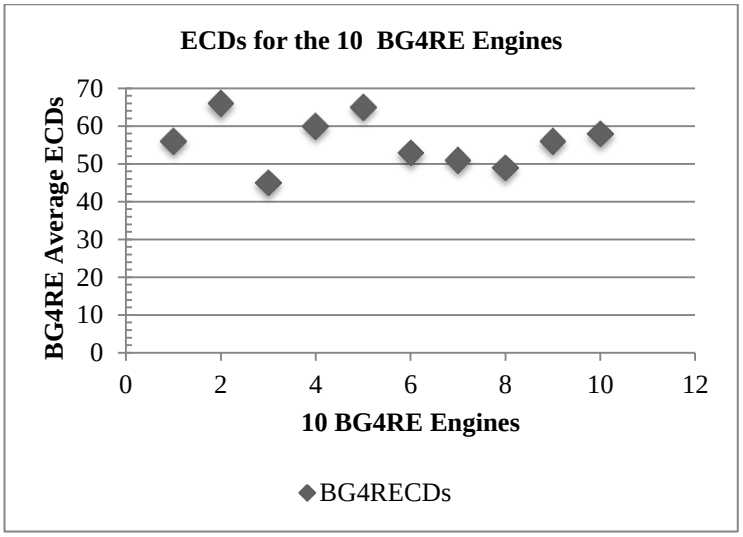


Figure 6.7: Distribution of BG4RE average ECDs against the 10 BG4RE Engines.

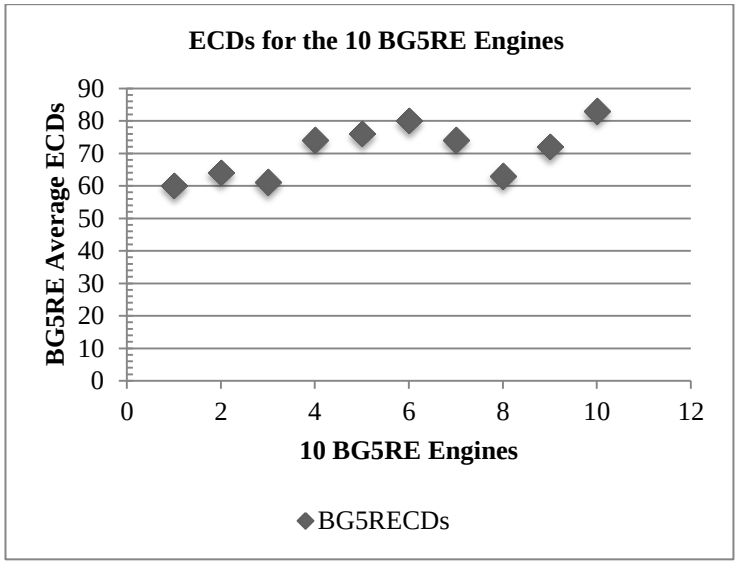


Figure 6.8: Distribution of BG5RE average ECDs against the 10 BG5RE engines.

ii. Graphs of the Distribution of ECDs and the Characters Matched

Figures 6.9 - 6.12 show the various graphs of the reported results as they appear in Table 6.2 and 6.3. However, this time the graphs show the number of ECDs and the total number of characters matched against each sub-REMEs, with n = 2, 3, 4 and 5. The graph is plotted against each 10 BGnRE engines. From Figure 6.9 and 6.10, one can deduced that by increasing the total number of characters matched, the average number of ECDs for BG2RE and BG3RE engines only grows steadily.

Furthermore, the diagram for the BG4RE engines in Figure 6.11 showed that increasing the number of regexps matched takes a gradual toll on the design. This is especially the case when in the worst case scenario the selected regexps that are extracted (refer to Section 5.3.1a for details on the extraction process) all have wild cards and length restrictions beyond 1000. Figure 6.12 showed an improvement in the BG5RE engines, even though the number of regexps implemented in the design increased in comparison to the BG4RE engines. This steady growth is attributed to the effectiveness of

the design and its ability to generate compact and controllable automata. The design grows rather slowly and yet steadily when the number of regexps matched is increased as shown in Figure 6.12. The design does not grow drastically as with some of the related approaches.

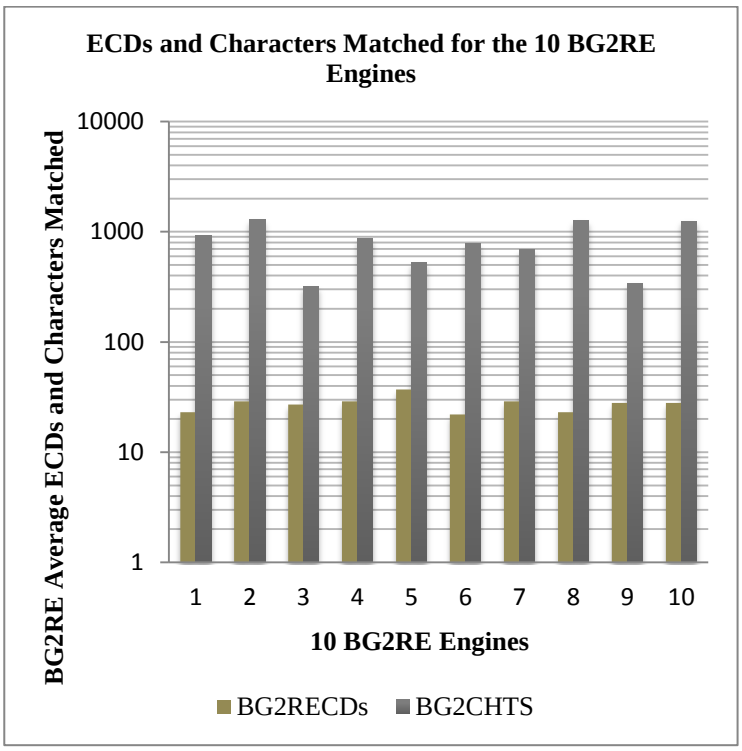


Figure 6.9: Graph for the ECDs and the total number of characters matched by the BG2RE engines.

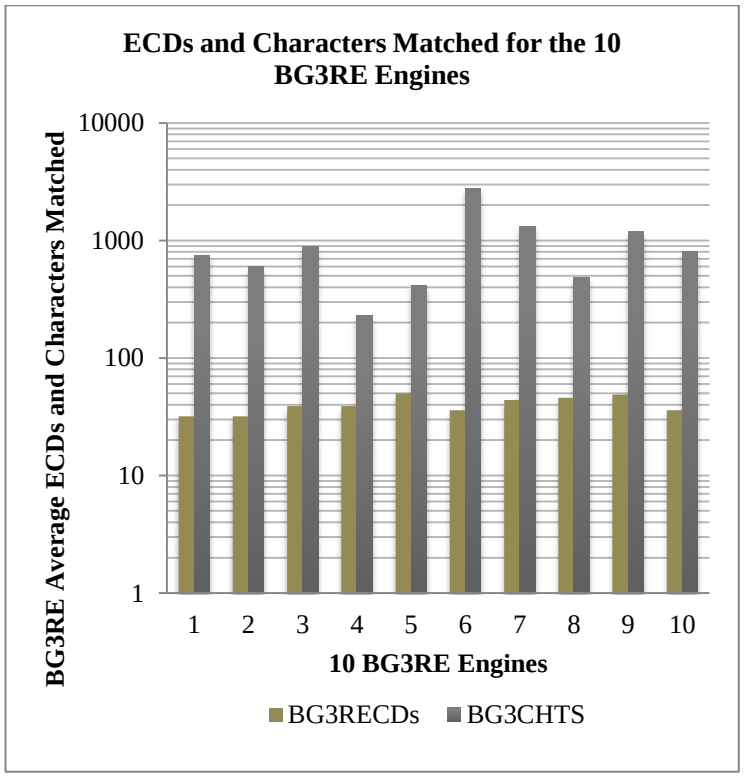


Figure 6.10: Graph for the ECDs and the total number of characters matched by the BG3RE engines.

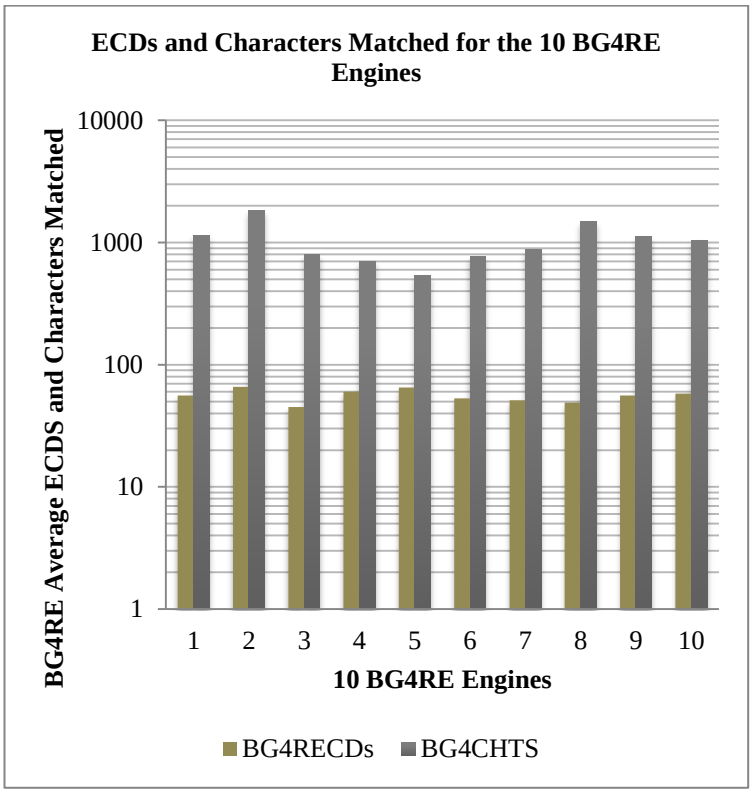


Figure 6.11: Graph for the ECDs and the total number of characters matched by the BG4RE engines.

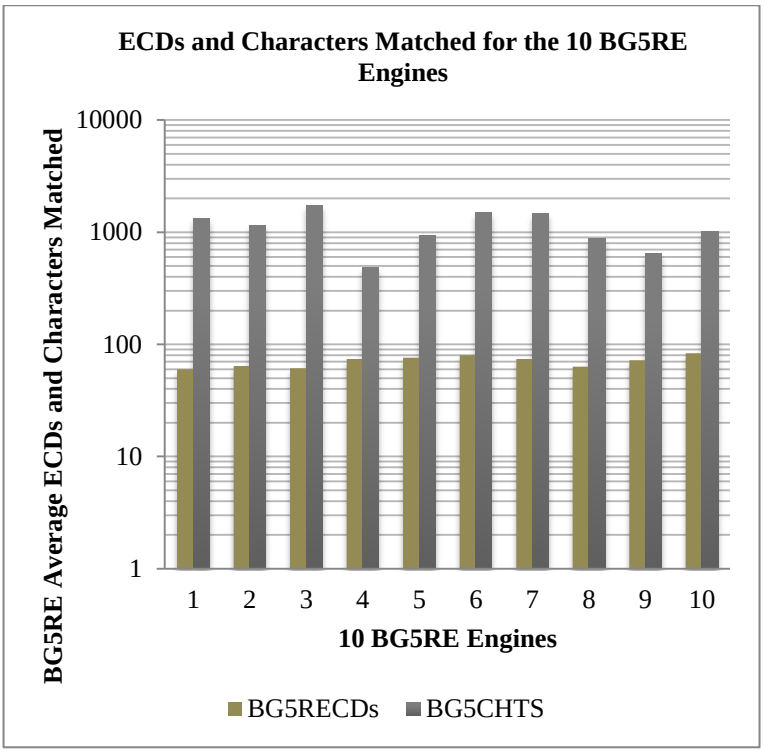


Figure 6.12: Graph for the ECDs and the total number of characters matched by BG5RE engines.

6.2.3 Analysis of the Synthesised ECD_RTS-NFA REME Designs

The complete tables of results have been generated for each of the acronym BGnRE synthesised designs for the BGnREs described in Section 6.2.2, where n = 2, 3, 4 and 5. The concerned designs are: BG2RE, BG3RE, BG4RE, and BG5RL. Each of engines is made up of 10 REMEs, with each REME made up of 4 sub-REMEs. This brings the total to 40 sub-REMEs for each BGnRE engine. The results of each category is summarised in Table 6.2 and Table 6.3. The detailed results are attached in Appendix 1.3 – 1.6 respectively for consideration.

a. BGnRE Schematic and Behavioural/Timing Simulation:

For the purpose of illustration, the design in Appendix 1.2 represent the RTL design view of a 4-byte BGnRE design, with n = 2. The design is synthesised into logic using the XST VHDL synthesis tool. The design represents the interfacing of the synthesised **RAMB36E1** (or simply 2x36kBRAMs) block through FFs. The FFs speed up the output of the 2x36kBRAM blocks (refer to Section 5.4.2a-i) the data is read from the BRAM blocks into the table-synthesis module (refer to Section 5.4.2a-ii) for performing the required table look up operations. Afterwards, a < 128-bit output is then generated and supplied into the four sub-NFA blocks (refer to Section 5.4.2a-iii). This is achieved through FFs as shown in Appendix 1.2.

From the schematic diagram in Appendix 1.2, the 4-bit MATCH port is the only output port. The ADDRESS, CLK, EN and SET_RESET ports are the only input ports of the test bench component specified in the component's instantiation process. The logic element 'ibuf'' is an input buffer, which connects the inputs to the ADDRESS, SET_RESET and EN nets. The logic element 'obuf'(output buffer) connects the MATCH output net. The logic element 'ibufg' is a dedicated buffer, with selectable I/O interface, and is used as the clock input. The buffer connects the CLK net, which is also connected to the global clock input AND gates (and4). The 'and4' gate is used for testing the condition of the current states. The 7-bit ECDs within the range of 0 - 127 trigger the necessary transitions to the next states within the sub-NFA block components. The remaining 128 – 256 inputs make up the redundant ECDs. Once a string of ECDs drive the transitions successfully from the initial state and is finally consumed at the accepting state, a 4-bit match output vector is obtained**.**

By way of illustration, Appendix 1.1 shows how a behavioural simulation (refer to Section 2.7 for more on simulation) is performed in one of the 10 sub-REME BG2RE engine designs that was selected. The ISE simulator (Isim) bundled in the Xilinx ISE FPGA Project Navigator, version 14.4 design suites is used to run the behavioural and timing simulation by supplying it with the necessary stimuli. The test bench file contains the stimuli data and is integrated within the entire VHDL design. Afterwards, the design is first simulated using the Isim simulator to confirm that the design is working according to specification. From the diagram in Appendix 1.1, the match variable in the diagram named m_match, has a match array value of '0001' at time 410ns as indicated by the purple coloured cursor as shown on the left-hand side of the diagram in Appendix 1.1. The match indicates that the first out of the four sub-REMEs found a match. The match is for the two regexps that were converted into the composite NFA of the first sub-REME block. Parallel matching occurs across the 10 sub-REME BG2RE engine designs synchronously.

b. Throughput and Throughput Efficiency

The results as shown in Appendix 1.3 – 1.6, Pages 142 - 143, details the individual REME design result as summarised in Table 6.4. Considering the BG2RE design first, the average number of LUTs (refer to Appendix 1.3) utilised by the design is 469, and the obtained average speed is 367.34MHz. The average number of states per character is 828, and the bus width is 32-bits. Only 2xRAMB36E1 memory per REME was required in the ECD_RTS-NFA, unlike the 20xRAMB36E1 memory per REME utilised by the initial ECD-NFA version. The throughput efficiency (Yang, Jiang and Prasanna 2008, p. 38) which is a standard method of computation is computed as follows:

a. Throughput (Tp) = speed (MHz) * 32 (data bus width in bits)/1024.

b. Throughput efficiency (Tpe) = (throughput * number of states)/number of LUTs).

= [(367.34*32)/1024]*828/469

= 11.48*1.7655

Tpe = **20.27**.

The Tpe of the BG2RE is 20.27. The same method is used to compute the throughput and throughput efficiencies of the BG3RE, BG4RE and BG5RE engine designs as shown in Table 6.4. The detailed result of each REME engine is as presented in Appendix 1.3 – 1.6.

c. Comparison of LUT-based REME Designs

Table 6.4 compares the results computed for the most related 4-byte LUT-based REME approaches with those of the BGnRE designs, with n = 2, 3, 4 and 5. Such designs reported the total number of LUTs utilised by their designs. The method of computing the throughput efficiency is as discussed in Section 6.2.3a.

Table 6.4: Compared results for related 4-byte LUT-based REME designs.

Design Approach	Input (bytes)	Speed (MHz)	Tp	NoCH	Tpe	NoL/ NoS
Yang, Jiang and Prasanna (2008)	4	233.13	7.46	15000	3.4	2.2
Yang and Prasanna (2012)	4	198.6	6.36	120000	9.09	0.7
Yang and Prasanna (2012)	4	166.7	5.33	100000	3.63	1.47
Yamagaki, Sidhu and Kamiya (2008)	4	113.4	3.63	40896	3.86	0.94
Clark and Schimmel (2004)	4	218.9	7	17537	2.3	3.1
BG2RE	4	367.34	11.44	8274	20.27	0.566
BG3RE	4	308.04	9.63	10229	8.39	1.489
BG3RE	4	293.71	9.18	10287	6.09	1.506
BG4RE	4	264.32	8.26	11127	3.54	2.332

The newly added column headings are: number of characters (**NoCH)**, number of LUTs **(NoL)**/number of states **(NoS)**. The last added column is **Tpe** = throughput * (number of states/number of LUTs). The **Tpe** refers to the throughput efficiency. The graph of the results as seen in Table 6.4 is as shown in

Figure 6.13 – 6.16. The LUT usage is the basis for computing the throughput efficiency, and is used to determine the overall efficiency of each separate approach.

Figure 6.13 shows that BG2RE REME design is by far having the highest throughput efficiency value, compared to any of the existing related LUT-based REME approaches. It has a Tpe value of 20.27, which is closely followed by the first design implemented by Yang and Prassana (2012) which has the value of 9.09. However, the other designs namely: BG3RE, BG4RE and lastly BG5RE engine with each having the value of 8.39, 6.09 and 3.54 respectively are better than other 4 remaining designs as shown in Figure 6.13.

Figure 6.14 is a confirmation of the performance of the BG2RE, BG3RE, BG4RE and BG5RE engine designs in terms of the respective obtained throughputs of the designs. When compared to the other LUT-based REME NFA designs. However, as the number of regexps per each sub-REME of the four categories of REMEs increased from 2 – 5 regexps, the designs witnessed a steady decline in the throughput as expected.

The ratio of LUT utilisation per each state of the automata implemented by the other related approaches is shown in Figure 6.15. From the figure, it is clear that the BG2RE engine design has the lowest ratio of LUTs per state utilisation compared to all the remaining designs. The BG3RE and BG4RE engines competed favourably with the remaining designs. Although the BG5RE engine performed better than two other designs, the 5-regexp implemented in the design took its toll on the design. However, it is expected that as more regexps are implemented within each design, the logic utilisation will tend to be higher. But, a steady growth rate of the logic utilised by any of the designs show that such a design can scale up even more with further optimisation.

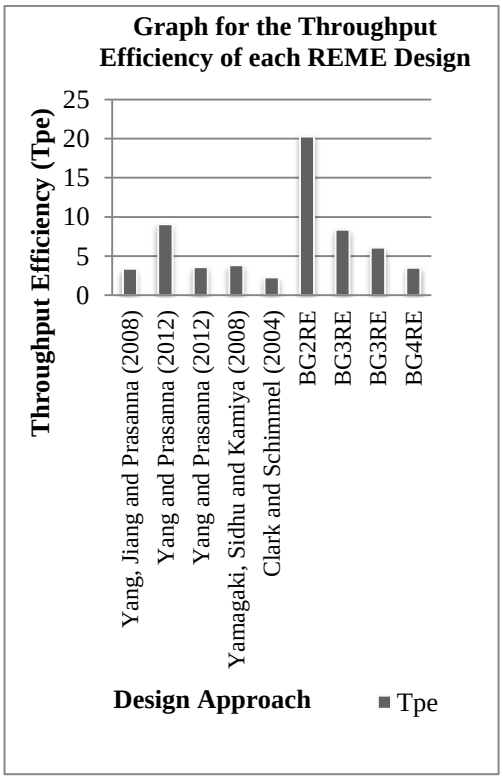


Figure 6.13: Graph for the throughput efficiency.

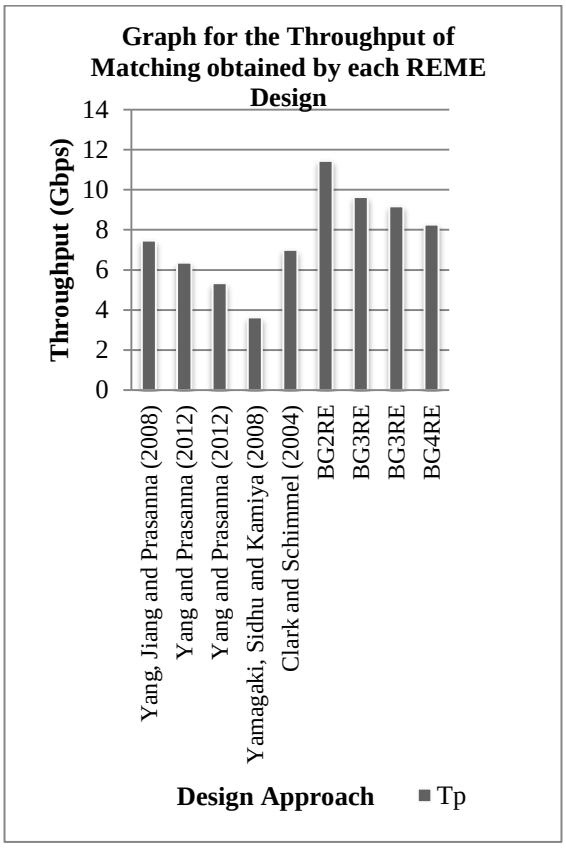


Figure 6.14: Graph for the throughput (Gbps).

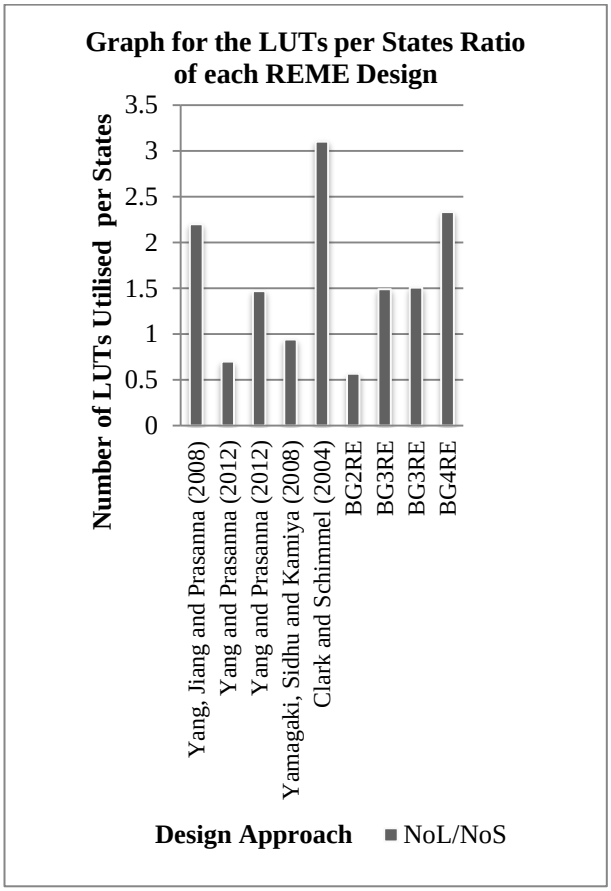


Figure 6.15: Graph for the number of LUTs per number of states utilised.

The BG2RE, BG3RE, BG4RE and BG5RE engine designs all recorded the highest speed of matching compared to all the other approaches as shown in Figure 6.16. The figure shows that the four REME designs sustained a much higher speed compared to the rest of the designs. This is bearing in mind that the level of complexity of all the separate designs are high. The least among the designs with the speed of 264.32MHz is still about 9.11% higher than the other related designs. The highest among them with 367.34MHz is about 51.62% higher than all the others.

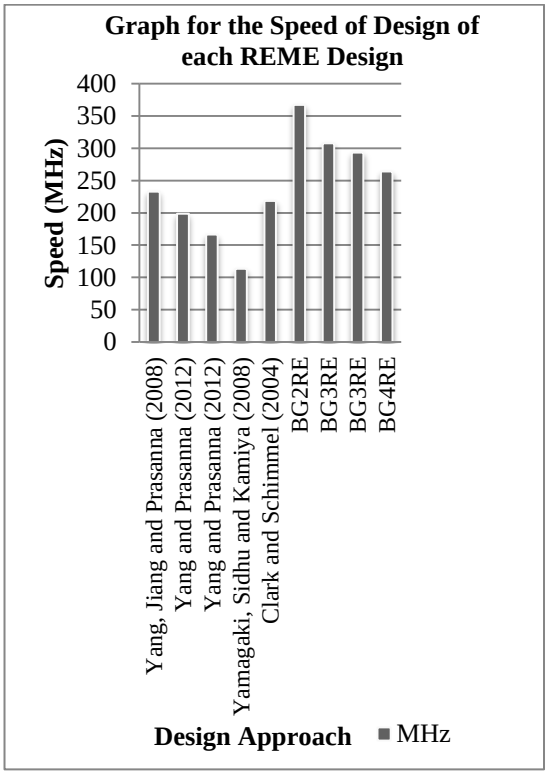


Figure 6.16: Graph for the speed (MHz) of matching.

Section 6.3 is dedicated for testing the research hypothesis which was stated earlier in Section 1.6, page 6. A statistical test that compares the means for all the four categories of designs namely: BG2RE, BG3RE, BG4RE and BG5RE engine, is utilised to test the results of experiments recorded as seen in Appendix 1.7 and 1.8, Page 144.

6.3 Test of Statistical Hypothesis

It is important to state that the results of the experiments for the 10 REMEs per each BGnRE design, where n = 2, 3, 4 and 5 were independently performed. As such the possibility that the reported estimates are based solely on chance is considered as the null hypothesis (H_0). The compliment of the null hypothesis is considered as the alternative hypothesis (H_1). The chosen significance error level is denoted by α (alpha), and a value of 0.05 is assigned to α in this test. That means up to 5% error is tolerated which is generally considered as the standard.

The statistic used in comparing the means of the group of REMEs is the One-way Analysis of Variance (One-way ANOVA) (IBM SPSS 2012). The ANOVA is used to divide the changeability among 3 or more groups as well as within groups. The changeability within groups is estimated as the sum of squares of the difference between each value and the mean value of the group. The changeability among groups is estimated as the difference between the mean of the various groups and that of the mean of all

the values in all the groups. A degree of freedom is associated with each sum of squares, and is usually computed based on the number of groups and number of elements within each group. A mean square value is computed by dividing each sum of squares by its degree of freedom (IBM SPSS 2012). The result of such a division is considered to be the variance. The F ratio which is used for testing equality of group means is computed as follows:

$$F = \frac{\text{Mean Square Between}}{\text{Mean Square Within}} = \frac{BSSM}{WSSM}$$

There is a probability value called the p-value that is computed from the values of the F ratio and the two degrees of freedom (df) values as shown in the One-way ANOVA Table 6.5 and Table 6.6. Whenever the p value is strictly less than α (i.e. $p < \alpha$), the test is said to be significant.

Also, there is a standard F distribution value denoted by F_{tab}, which has been precompiled and tabulated in the standard F distribution table reported by Rohatgi and Saleh (2001, pp. 681-683). The value of F_{tab} is obtained from the table using the numerator and denominator degrees of freedom that were pre-computed using the One-way ANOVA. If the tabulated value F_{tab} is strictly less than the calculated F ratio denoted by F_{cal} using the One-way ANOVA (i.e $F_{tab} < F_{cal}$), then the null hypothesis H_0 is rejected. This is because a large F ratio means that the variability among groups cannot happen by chance alone.

A One-way ANOVA analysis was then performed on the results of experiment as shown Appendix 1.7 and 1.8. Appendix 1.7 contains the average throughput values for the 10 REMEs per each BGnRE design, where n = 2, 3, 4 and 5 grouped together. Appendix 1.8 contains the results for the number of LUTs utilised by each state contained in the same BGnRE engines. The throughput efficiency of the BGnRE engine designs was computed based on the average throughput and the LUTs utilised by each state of the engines as shown in Figure 6.13 and also discussed in Section 6.2. To perform the test, the Statistical Package for the Social Sciences (SPSS) version 21 provided by IBM SPSS (2012) was used.

6.3.1 Testing the Throughput

The One-way ANOVA analysis was performed for average throughput values of the 10 REMEs per each BGnRE design, where n = 2, 3, 4 and 5. The REMEs were grouped together as seen in Appendix 1.7, while the analysis is as shown in Table 6.5. The F_{tab} value obtained from the standard F distribution table reported by Rohatgi and Saleh (2001, pp. 681-683) using the two degrees of freedom values 3 and 36 is as shown Table 6.5. The value obtained is $F_{tab} = 2.86$. This clearly shows that with $F_{cal} = 20.823$, then with $F_{tab} < F_{cal}$, as such the null hypothesis (H_0) is rejected. Also, the chosen significance error level is $\alpha = 0.05$ while $p = 0.000$ as shown in Table 6.5. This shows that with $p < \alpha$, the test is significant.

Table 6.5: One-way ANOVA Analysis for the average throughputs of REMEs.

BGnREs	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	54.916	3	18.305	20.823	.000
Within Groups	31.648	36	.879		
Total	86.564	39			

6.3.2 Testing the Ratio of LUTs/States

The One-way ANOVA analysis was also performed for the ratio of LUTs utilised by each state of the automata of the 10 REMEs per each BGnRE design, where n = 2, 3, 4 and 5. Given that F_{tab} = 2.86 (Rohatgi and Saleh 2001, pp. 681-683), and that α = 0.05. The REMEs were also grouped together as seen in Appendix 1.8, while the test is as shown in Table 6.6. It can be clearly seen from Table 6.6 that F_{cal} = 4.492 and p = 0.009. As such, because $F_{tab} < F_{cal}$, the null hypothesis H_0 is rejected. And because $p < \alpha$, the test is significant.

Table 6.6: One-way ANOVA Analysis for LUTs/States of REMEs.

BGnRE	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	19.734	3	6.578	4.492	.009
Within Groups	52.712	36	1.464		
Total	72.445	39			

Based on the ANOVA analysis as discussed in Section 6.3.1 and 6.3.2, the conclusion is that we shall reject the null hypothesis H_0 and accept the alternative hypothesis H_1. This is because by increasing the number of regexps in each sub-REME contained in the 10 BGnRE design, where n = 2, 3, 4 and 5, the throughput and the number of LUTs utilised by each sub-REME in the overall REME is affected. This in turn has an effect on the overall throughput efficiency. However, the decline in the throughput and throughput efficiency experienced across the four groups of REMEs is steady and not drastic. This shows some promise that the design can actually be sustained and scaled up even beyond the BG5RE REME design, provided that the ECD_RTS-NFA approach can be further optimised.

6.4 Chapter Summary

The ECD_RTS-NFA approach showed that the BGnRE design, where n = 2, 3, 4 and 5 performed as expected. The BG2RE recorded the highest throughput and throughput efficiency compared to all the other related designs. Overall the chapter showed that it is possible to efficiently scale up the design beyond the BG5RE engine design. Although as the number of regexps is increased within each of the implemented REMEs, the throughput and the throughput efficiency declines. However, the decline is steady and not rapid as expected, which shows some promise for the ECD_RTS-NFA approach.

The ECD_RTS-NFA approach is however affected especially by regexps consisting of longer length restrictions beyond a 1000. Such regexps easily increase the number of LUTs utilised per each state of the NFAs placed within each REME group. Notwithstanding, each REME design ended up utilising 2x36kBRAM memories for every 4 sub-REMEs. For instance, the BG5RE engine design of the ECD_RTS-NFA approach utilised only a 2x36kBRAM to represent 20 regexps. This reflects a 90% reduction in memory requirement compared to the initial ECD-NFA approach as a result of optimisation.

Moreover, the important benefits of the approach in this thesis is its ability to build stable, efficient and multi-pattern REMEs arranged in parallel to perform pattern matching. The approach achieves at least 75% reduction in the total number of REMEs, compared to the ones described by Mitra, Najjar and Bhuyan (2007), Yang, Jiang and Prasanna (2008), Yang and Prasanna (2009), and Ganegedara, Yang and Prasanna (2010) when considering less complex patterns. While about 30% reduction is achieved when considering more complex patterns. A Dell personal computer (PC) system running on A 64-bit windows

7 OS platform was used to implement the ECD_RTS-NFA REME designs. The PC is fitted with a 2.67GHz CPU and 8.00GB installed RAM. The REME designs utilised minimal logic resources compared to most of the other related approaches. This is evident in the reported values of the throughput efficiencies shown in Appendix 1.3 – 1.6 and summarised in Table 6.4. Chapter 7 discusses the conclusions drawn from the thesis

7. Conclusions

The aim of this thesis was to efficiently create an input classification-based based approach. The approach was based on the concept of Equivalence Classification that was implemented for a target FPGA platform. The classification process was able to generate a categorised set of inputs referred to as Equivalence Class Descriptors (ECDs). The ECDs were used to drive the various NFA-based regexp pattern matching engines called REMEs in the initial approach called the ECD-NFA. After further optimisations, the ECD-NFA approach was then transformed into a more new version called the ECD_RTS-NFA. The ECD_RTS-NFA approach is made up of a set of more efficient parallel matching REMEs. The REMEs in each of the four categories of ECD_RTS-NFA designs combined multiple optimisations into a single design. The overall design was broadly divided into two separate but interleaved phases which represent the software (parser) and hardware (target FPGA implementation) phases.

7.1 Contributions and Conclusions

A novel ECD-NFA two-phased and toolchain-based approach was designed and implemented. After a number of optimisations, the initial design was transformed into an ECD_RTS-NFA design. The design generated efficient NFAs for the various parallel matching REMEs. This was achieved through the simple and less complex concept of equivalence classification. The concept was capable of classifying the various inputs on the automata. The classified inputs generated what is referred to as equivalence class descriptors, or simply ECDs. The ECDs were then used to represent the various classes of the compressed inputs based on their effect on the NFA-based automata. Each ECD represented state vectors, and were composed of sets of vectors of next states (refer to Section 3.2.2f-i and 5.3.3 for more details).

The approach consisted of functional individual and collective BRAM blocks. The way the various BRAM blocks interfaced with the table-synthesis blocks as well as the NFA blocks was explained in Chapter 5 and analysed in Chapter 6. The original ECD-NFA design experienced some optimisation issues (refer to Section 5.3.3). The issues led to development of the optimised version, the ECD_RTS-NFA. The issues include the following:

a. Synthesis, place and route (PAR) processing delay: It was observed that the unnecessary time it took to synthesise and finally PAR the design was in the magnitude of 8 hours or more. This was predominantly caused by the complexity of the initial decoding module contained in each REME pipeline (refer to Section 5.3.1b). The affected module is the module labelled 'T_c' (compressed-

2D table of ECDs in the 4xDecoder units) as shown in Figure 5.1. The same module is also shown in Figure 5.3 and labelled as '4xECDs Decoder Module'.

b. Use of nested loops: Decoders with 128-bit wide shift registers were used to perform a table look up operation in the decoding module described in (a). The process initially required to fetch ECDs from the BRAM modules as seen in Figure 5.1 and 5.3. The BRAM module that was involved and labelled as 4x256*8-bits) BRAM module, released the requested ECDs to the decoding modules described in (a). After process by the decoding modules is completed, bit vectors of outputs were generated. The outputs were then passed to the ECD-NFA modules, labelled as 4 x ECD-NFA module (refer to Section 5.4.2a).

   However, the main problem was that the decoding process in the decoding module (refer to Section 5.4.2a-ii) required the use of a 4-level nested loop to perform the 2-byte and 4-byte look up operations involved (refer to Algorithm 5.2). The process took too long to synthesise even the most trivial tables of ECDs. In some cases the synthesis process just failed eventually. This was attributed to the fact that, the FPGA platform was never designed to perform multi-level nested loop operations, unlike in most high level languages such as Java. Also, the amount of logic needed to perform such operations was enormous. In fact, the process required that the entire logic circuits contained in the circuit elaboration should be fully constructed first. Afterwards, the time consuming process of trimming out all the irrelevant logic components begins. The XST achieves this by employing some clever in-built optimisation strategies during synthesis.

c. Use of excess resources: Eventually it turned out that on occasions where the synthesis of the decoding module described in (b) was successfully converted into logic, the amount of LUTs, shift registers, and other logic components was simply too high. This significant logic circuit size, translates to poor throughput efficiency. This is inefficient for determining the performance of all the REME related approaches including the one in this thesis.

The contributions made to this thesis have already been highlighted in Section 1.7. As a quick reminder, the contributions made can be summarised thus:

i. This thesis introduced a novel ECD-NFA two-phased and toolchain approach (refer to Section 3.2.2f-i) with its optimised version called the ECD_RTS-NFA. The approach was used to generate efficient NFAs for the various REMEs. The REMEs were then arranged in parallel to perform multi-character and multi-pattern matching. This was made possible through the use of the simple and less complex concept of equivalence classification.

   The process classified the various inputs on the automata, and then generated the relevant ECDs. The ECDs are then used to represent the various classes of the compressed inputs based on their effect on the NFA-based automata. This was a deviation from the DFA based schemes described by Brodie, Taylor and Cytron (2006), and Tripp (2008, p. 4). The process described in the approach performed alphabet reduction in the process. Section 3.2.2f also explained the alphabet reduction compression process described by some classification-based approaches.

ii. The process involved in performing (i) uniquely applies to NFAs only. This is because with NFAs, there is not necessarily a single current state involved anymore, unlike with DFAs. Another reason is that given a single ECD input, several states could become active at once.

In order to implement the classification-based scheme, a type of state transition table that determined the set of next states for each active current state, was defined. Members of an equivalence class were identified as the ones that had identical columns in the table. However, on this occasion, the table columns could no longer be columns of next states anymore. The table columns are now vectors of next states.

iii. The redundancies found within the ECDs were further exploited, by applying an efficient compression technique. Also, the algorithm was able to eliminate all self and empty string transitions, thereby performing edge reduction in the process. This helped us to reduce the number of redundant transitions on the automata. Section 3.2.1b - 3.2.1d explains more on edge (transition) reduction schemes.

iv. Once the ECD-NFAs are generated, a minimised and compressed n-dimensional table of compressed ECDs (Gupta and McKeown 1999, p. 150) was created, suitable for a multi-byte input match. The multi-byte matching process was necessary for increasing the stride (Brodie, Taylor and Cytron 2006) of matching. The process was then applied recursively to create an n-byte matching process, with $n = 2^k$, where $k = 1$ and 2.

v. A simple algorithm was later designed and implemented to synthesise the ECDs tables generated in (iv) into logic. This helped to significantly cut down a lot of irrelevant logic resources such as: shift registers and decoders, which were a major bottleneck to the ECD-NFA design. However, the bottlenecks were completely eliminated in the optimised ECD_RTS-NFA design.

vi. A very simple and less complex toolchain for implementing simple, fast and area efficient NFA-based REMEs (Mitra, Najjar and Bhuyan 2007; Yang, Jiang and Prasanna 2008; Yang and Prasanna 2009; Ganegedara, Yang and Prasanna 2010) was developed. The toolchain was divided into two phases: first phase (software implementation) and the second phase (hardware implementation on FPGAs).

vii. The two-phased approach comprehensively utilised the techniques which combined multiple optimisations such as: edge reduction, alphabet reduction, increased striding (Becchi and Crowley 2008, p. 50), input classification (Brodie, Taylor and Cytron 2006; Tripp 2006; Arnold 2007), infix, prefix and suffix sharing (Hutchings, Franklin and Carver 2002; Sourdis and Pnevmatikatos 2004; Yu et al. 2006; Lee et al. 2007; Lin et al. 2006). The approach also used input classification (Brodie, Taylor and Cytron 2006; Gupta and McKeown 1999, p. 150), to create compact memory efficient and fast NFAs for the hardware-based regexp matching REMEs. The techniques mentioned were discussed in Chapter 3.

viii. The approach also performed multi-character matching (Sourdis and Pnevmatikatos 2004; Becchi and Crowley 2008; Clark and Schimmel 2004; Jiang, Yang and Prasanna 2010) at high speed, built into the REMEs (Mitra, Najjar and Bhuyan 2007; Yang, Jiang and Prasanna 2008; Yang and Prasanna 2009; Ganegedara, Yang and Prasanna 2010). The approach was designed to optimally use the limited available FPGA LUTs and other logic circuits.

ix. A technique further was implemented that built nested sub-REME blocks into each REME mentioned in (vi). The blocks were then arranged in parallel to perform multi-pattern matching. The matching process was designed to use fewer REMEs than the ones proposed and

implemented by Mitra, Najjar and Bhuyan (2007), Yang, Jiang and Prasanna (2008), Yang and Prasanna (2009), Ganegedara, Yang and Prasanna (2010).

x. Using the novel ECD-NFA optimised version referred to as ECD_RTS-NFA, a unique form of block RAM (BRAM) compression for the ECDs was utilised. The compressed BRAMs generated in the first phase as mentioned in (vi) are used to supply 7-bit ECDs to the various matching units. This is in contrast to the way the BRAM centralised character matching approach (Xilinx 2011; Yang, Jiang and Prasanna 2008; Yang and Prasanna 2009; Ganegedara, Yang and Prasanna 2010; Hieu et al. 2011; Long et al. 2011) was implemented for character matching.

xi. The required VHDL files representing the multi-character and multi-pattern REME blocks were efficiently and automatically generated. The files were then used to construct the required hardware REMEs in (ix), and served as input to the second phase in (vi).

xii. The major idea behind the approach was to create an all-round design that combined numerous approaches into a single approach. The design was capable of using equivalence classification for an NFA-based design, previously known to work with DFAs only as described by Brodie, Taylor and Cytron (2006, p. 194) and Tripp (2006). This was followed closely by implementing the optimisations mentioned in (vii).

7.2 Future Work

The future works outlined in Section 7.2.1 are based on the limitations of some of designs proposed in Chapter 3, and how they may be improved. Also the future work described in Section 7.2.1a relates to how the design could be extended. However, the design serves as a foundational platform for designs that are LUT-based and are constructed using the concept of equivalence classification.

7.2.1 Proposed Improvements

a. Length Restriction Problems

Section 2.3 has shown that the issue of state explosion attributed to complex regexp affects most conventional DFA and non-optimised NFA approaches. It is especially the case when patterns containing wildcards (.*) are combined to create composite DFAs as observed by Becchi and Crowley (2007b) and Yu et al. (2006). The condition becomes even worse, when such patterns contain significant length restrictions beyond a 1000 such as: .* a{1024,}e, abc.*cd{250}.*{250}.

Furthermore, the efficient and compact nature of the ECD_RTS-NFA design makes it stable and mostly unaffected by such state explosion issues. However, without some counting mechanism that is built upon a counter logic circuit, it is almost impossible to further reduce the amount of utilised logic circuit elements any further than it already is. Furthermore, as mentioned in Section 7c, the throughput efficiency is greatly affected by the overall logic circuit size for the circuit implementation of the design. Moreover, the bigger the logic circuit size, the poorer the overall design speed and the overall throughput of matching.

A counter-based mechanism is by all means required. The mechanism needs to be constructed upon the ECD_RTS-NFA approach to extend it further. Also, there is a significant amount of length restrictions prevalent in most of the current Snort rules, which easily increases the complexity of the design automata (Becchi and Crowley 2007b; Aycock 2006) during implementation. However, the

problem could be brought under control, but the difficulty lies in how best to achieve that without incurring more logic circuit costs during the hardware implementation. The counter mechanism should be able to keep track of the number of repetitions against the matched characters that are constantly streamed through the network. This could be achieved by utilising some simple shift registers and FFs combined in a particular way. Afterwards, the perceived mechanism should be implemented in the parser to form a counter-based module, and then finally generated as part of the VHDL files that represents the entire design.

However, the counter mechanism has to ensure that the automata states remain unchanged while the counting is performed. To achieve this, some extra memories will be required to effectively keep track of which characters were matched repeatedly, and the instances that the characters were matched. The problem is that there is no FPGA device primitive memory that exists with such a capacity? And even if one exists, the key question is can it fit into the FPGA architecture and still be made portable enough to store every character streaming through the network?

Notwithstanding, the ECD_RTS-NFA approach can serve as a platform for any future counter-based designs. This is because no new states will be formed during the process of integrating such a mechanism, as the issue of state explosion has been brought under control by the design. Furthermore, successfully building and integrating such a counter-based mechanism into the ECD_RTS-NFA design, will create a design that is faster, compact, memory efficient, and more scalable. The design will also experience higher throughput and throughput efficiency too. But all this is easier said than done.

b. Cycle-Based Problems

The design by Lin, Tai and Chang (2007) showed how to reduce the total number of states and transitions present on a given naive FSM. This was achieved by merging appropriate states and eliminating unnecessary transitions. However, the problem is further compounded when multiple patterns are added to the existing graph, leading to cycle problems. The cycle problems lead to false positive matching in the final merged FSM (refer to Section 3.2.1b-i for more details).

In order to solve such problems, the addition of an extra buffer is proposed. The buffer is to be added to the table that stores the information regarding the next character to be matched from any given current state. For instance, state 5 as seen in the AC FSM of Figure 3.6 led to the formation of a cycle. A test should be performed at that state to test if the next character to be consumed in the pattern is not an ‘f’ in order to complete the pattern match for the string pattern “abcdef”. If it turns out to be a ‘b’ instead of an ‘f’, forcing the AC FSM to start a match for the false positive string “abcdebcdef” instead, then the following should happen to resolve the problem:

i. Split the pattern into four, starting from state 5. This is because state 5 is common to both patterns “abcdef” and “wdebcg”. This will form the sub-patterns “abcde” as pattern1, “wde” as pattern2, “f” as pattern3 and “cg” as pattern4.
ii. There is the need to add additional state i from state 5 on input of b to another state t. From state t we can then transit to another added state v on input of c. From state v we can then transit to the accepting state 8 on the input of g. The former transition from state 3, which fell within the cycle on input of g can then be cut off. This will ensure that no cycle remains again on the AC FSM.
iii. Lastly, the same pathVec and ifFinal vectors shall be utilised, together with the buffer, created during the merging of states as explained in Section 3.2.1b-i. A failure will then be reported

immediately if pattern3 does not begin with a character ‘f’. Also a failure will be reported if pattern4 does not begin with character ‘c’ and followed by ‘g’. This will force pattern1 and pattern2 to terminate by reporting a 0 at the lsb position of their respective match vectors.

This comes with an additional cost of two extra transitions and an additional state between states 5 and 8. However, the benefit is that no cycle will be formed. The idea of pattern splitting was also implemented by Kumar et al. (2006) in their Delayed Input DFAs (D^2FA) approach in Section 3.2.1b-ii.

c. Memory Centric Based Problems

With BRAM centralised character based matching approaches such as the one by Yang, Jiang and Prasanna (2008), Yang and Prasanna (2009), Ganegedara, Yang and Prasanna (2010), Becchi and Crowley (2008), Hieu et al. (2011), and Long et al. (2011), the basic problem can be deduced as follows:

i. Use of 256-bits: Storing each character class of inputs as a 256x72-bit column on BRAMs is wasteful. This is especially true when only a single character such as ‘a*’* is stored as an input, which takes up 256-bits of space alone. This is because the bit position of the character ‘*a’* is a 1, while all [^a] are represented by 0’s. In the approach described in this thesis, just 7 bits was used to represent each of the ECDs, and achieved the same purpose.

ii. Complexity of patterns: The complexity of patterns determines the number of inputs generated when performing increased striding such as: 2-byte, and 4-byte matching. It then implies that the number of columns of the BRAMs used in most of the approaches mentioned earlier is very likely to exceed the maximum 72 columns for a 256x72-bit arrangement. That means that there will be additional requirement for additional primitive memories.

However, with the classification approach described in this thesis, the growth of the input size has been successfully contained. This is because the design creates at most < 128 ECDs for every REME block. It then follows that by simply instantiating four 256x8-bit BRAM blocks for every sub-REME within a REME Block, every input will be represented as shown in Figure 5.9a.

d. Evaluation

The Personal Computer (PC) used for benchmarking in the evaluation of this thesis is made up of an Intel-based processor. The PC is fitted with 8GB of installed primary memory. It has an observed 3700MB of usable memory, and 3913 cached memory. The PC has a maximum clock operating frequency of 2.67GHz. Each category of design namely: BG2RE, BG3RE, BG4RE, and BG5RE engines are versions of the ECD_RTS-NFA design. Furthermore, each design was built using extracted regexps from the VRT Rule distributed by Sourcefire (v 2.0) community rules, 2001-2013 (Snort 2013; Sourcefire 2009) provided by Snort. The regexps were obtained from Snort NIDS and evaluated accordingly in Chapter 6.

Furthermore, with a lower number of regexps compiled per REME, particularly in the case of the BG2RE design, the expected throughput of the design was the highest compared to all the other 4-byte related matching designs. The BG2RE design reported a 14% improvement over the next best reported throughput, and a 93% improvement over the worst reported throughput as seen in Table 6.1.

Secondly, in terms of throughput efficiency the BG2RE engine reported over 58.61% improvement over the next best throughput efficiency belonging to the BG3RE engine. The design also

recorded an 88.65% improvement over the worst related approach as seen in Table 6.1. Each of the ECD_RTS-NFA REME designs only utilised a 2x36kBRAM memory blocks. For instance, the BG5RE design utilised 2x36kBRAM rather than 20x36kBRAM memory blocks. This reduction portrayed a 90% reduction in the total number of required primitive memories. The throughput and throughput efficiency only steadily declined with increased pattern complexity as stated in the research hypothesis.

7.3 Chapter Summary

An ECD_RTS-NFA design that could improve throughput with sustained throughput efficiency was proposed and implemented. The design was optimised to improve the synthesis and PAR time, by eliminating the use of decoders initially synthesised to perform table look up operations. The design also eliminated the use of nested loop operations usually associated with such decoding operations. This is important as FPGA devices are not designed to perform nested loop operations. The ECD_RTS-NFA performs multi-character matching like the other related approaches, but in addition it also performs multi-pattern matching all within each REME design. The level of complexity in such a case is not comparable to the other related approaches, even though the parallel matching pipelined arrangement may be likened.

The equivalence classification concept which was used to generate inputs (ECDs) is only applicable to NFAs. This is because with NFAs, there is not necessarily a single current state involved anymore, as any classes of ECDs can activate several transitions and states at once. The implemented NFA minimisation technique helped in identifying, classifying and merging all the vectors of next states. The vectors were each efficiently and automatically assigned a unique ECD. Each ECD represents hundreds of separate similar state input characters that would have ordinarily been stored on the state transition table, which would have been impossible to efficiently store. This is especially important, when increasing the stride of matching by consuming multiple characters at once.

Another important contribution made by the ECD_RTS-NFA approach is that it reduced memory wastage. This was achieved by utilising only 2x36kBRAMs to implement REMEs consisting of up to 20 regexps of varying lengths. This is an improvement over the initial 20x36kBRAMs for the same 20 regexps utilised by the ECD-NFA and other related approaches. The improvement in the design is measured in terms of the throughput of matching and the number of logic resources utilised within a confined circuit area in the target FPGA. The ECD_RTS-NFA REMEs for each level of scaled REME designs are: BG2RE, BG3RE, BG4RE, and BG5RE engines. The BG2RE particularly recorded the best performance compared to the other groups of REMEs and the other related approaches as seen in Table 6.1. Furthermore, due to the pipelined arrangement of the REMEs, the whole process of performing table look up operation became fast and efficient. Also, due to the compact nature of the table-synthesis approach, adding more patterns to each REME only had a linear declining effect on the overall design and the size of the ECD tables. As such, the design has a very good prospect of scaling up steadily with every step-wise increase in the number of compiled regexp per each sub-REME. This is also an important contribution to this thesis.

Furthermore, the ECD_RTS-NFA approach does not utilise the centralised BRAM character classification technique implemented by some of the other related. The BRAM character classification scheme was used by some approaches to supply character inputs to their various design REMEs, which is not too memory efficient. The ECD_RTS-NFA approach instead utilises the novel compressed ECD table

synthesis technique. The technique helped to share up to 4-bytes of classified ECDs to the various REMEs efficiently. The compressed table of ECDs has an advantage over the centralised BRAM classification scheme. This because, the compressed table of ECDs scheme guarantees that each ECD is represented using just 7-bits instead of the 256-bits utilised by such centralised BRAM classification approaches.

However, an incremental design which can serve as alternative approach and capable of building upon the existing ECD_RTS-NFA design platform is required in future. The current platform generates scalable and stable NFA REMEs. But the design is in need of a counting mechanism to further minimise the number of logic resources used when synthesising complex REMEs consisting of regexps with long length restrictions beyond a 1000. With a counter-based ECD_RTS-NFA approach that extends the current approach, the steady decline in the number of logic circuits utilised could be curtailed even further. Such a growth becomes evident as the regexp REMEs begin to exceed the threshold of n>5 in the BGnRE engines, where n = 2, 3, 4, 5. The proposed counter-based ECD_RTS-NFA design can improve the current design, and make it more efficient in terms of performance. Also, with such a proposed design, fewer REMEs will be required to match thousands of regexp patterns. A minimal logic circuit cost in terms of LUT utilisation and other logic circuits, with a more sustained decline in the throughput of matching was shown to occur during implementation. This is also bearing in mind that the throughput efficiency is almost guaranteed to decline at a steady rate, and not drastically.

7.4 Concluding Thoughts

The future of FPGA-based designs such as the ECD_RTS-NFA is bright. This is because there are on-going researches that are trying to improve even the most fundamental building blocks of digital designs such as the D-type Flip-Flop (DFF). The DFF according to Trefzer et al. (2015) is a widely utilised logic circuit for 'pipelining in digital signal processing (DSP) and register files in microprocessors'. Timing is essential in achieving maximum achievable clock speeds by most FPGA-based REME designs. This is especially the case with implemented approaches such as the ECD_RTS-NFA. As such, an improvement in the clock-to-q delay, setup time, hold time and dynamic power consumption of DFF (Trefzer et al. 2015, p. 192) will have tremendous impact on the overall throughput of matching in any given REME circuit. FPGAs also provide high performance per watt of power consumption (Lacey, Taylor and Areibi 2016), making them flexible and power efficient in industrial application.

High performance, greater flexibility, lower development cost and the faster time-to-market is attributed to current FPGAs. This gives most NIDS especially signature-based NIDS requiring parallel computations a genuine advantage when implemented using FPGAs. Also, by combining optimisations such as multi-character matching and BRAM-based character classification techniques (Singapura et al. 2015), the FPGA can be exploited even further. Furthermore, mainstream software development practices show that FPGAs are attractive choices. This is because the FPGA tools have since adopted software-level programming models including the open parallel programming models (OpenCL) standard. Also Lacey, Taylor and Areibi (2016) further explain that FPGAs are currently shifting towards System-on Chip (SoC), where Advanced Reduced Instruction Set Computer Machines (ARM) 32-bit and 64-bit processors and FPGAs are placed on the same fabric.

Currently, Programmable Logic Devices (PLDs) are highly sought after. This is further confirmed by the WinterGreen market report of 2010 (WinterGreen Research Inc. 2010). The report which has about

287 pages and 148 tables and figures show that the global market for PLDs such as FPGAs has risen from $3.5 billion in 2009 and is anticipated to reach $9.6 billion by the end of year 2016. As such, there is a future for FPGAs and for applications that can take full advantage of the advancements made in the development of modern FPGAs.

BIBLIOGRAPHY

Aho, A. V. and Corasick, M. J. (1975). Efficient String Matching: An aid to Bibliographic Search. Communications of the ACM, 18(6), 333-340. DOI: 10.1145/360825.360855.

Alicherry, M., Muthuprasanna, M. and Kumar, V. (2006). High Speed Pattern Matching for Network IDS/IPS. Proceedings of the 2006 IEEE International Conference on Network Protocols. Santa Barbara, CA: IEEE Computer Science Press, pp. 187 -196. DOI: 10.1109/ICNP.2006.320212.

Arnold, J. T. (2007). Lecture Notes for MATH 3034. An Introduction to Mathematical Proofs. Lecture Material edn. Blacksburg, Virginia: Department of Mathematics, pp. 1-13.

Aycock, J. (2006). Computer Viruses and Malware. Advances in Information Security. Vol. 1. USA: Springer. ISBN: 978-0-387-34188-0.

Bachrach, J., et al. (2012). Chisel: Constructing Hardware in Scala Embedded Language. Proceedings of the 49th Annual Design Automation Conference on - DAC '12. San Francisco, USA: ACM, pp. 1216-1225. DOI: 10.1145/2228360.2228584.

Bacon, D., Rabbah, R. and Shukla, S. (2013). FPGA Programming for Masses. , 11(2), 1-13. DOI: 10.1145/2436696.2443836.

Baker, Z. K. and Prasanna, V. K. (2004). A Methodology for Synthesis of Efficient Intrusion Detection. Systems on FPGAs. Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. Napa, CA, USA: IEEE, pp. 135 -144.DOI: 10.1109/FCCM.2004.6.

Becchi, M. and Cadambi, S. (2007). Memory-Efficient Regular Expression Search using State Merging. Proceedings of the IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications. Anchorage, AK: IEEE, pp. 1064 -1072. DOI: 10.1109/INFCOM.2007.128.

Becchi, M. and Crowley, P. (2008). Efficient Regular Expression Evaluation. Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems - ANCS '08. San Jose, CA, USA: ACM/IEEE, pp. 50-59. DOI: 10.1145/1477942.1477950.

Becchi, M. and Crowley, P. (2007b). A Hybrid Finite Automaton for Practical Deep Packet Inspection. Proceedings of the 2007 ACM CoNEXT Conference on - CoNEXT '07. New York, NY: ACM, pp. 1. DOI: 10.1145/1364654.1364656.

Becchi, M. and Crowley, P. (2007a). An Improved Algorithm to Accelerate Regular Expression Evaluation. Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems - ANCS '07. Orlando, Florida: ACM/IEEE, pp. 145-154. DOI: 10.1145/1323548.1323573.

Bispo, J., et al. (2007). Synthesis of Regular Expressions Targeting FPGAs: Current Status and Open Issues. In: Reconfigurable Computing: Architectures, Tools and Applications. Lecture Notes in Computer Science. Vol. 4419. Mangaratiba, Brazil: Springer Berlin Heidelberg, pp. 179-190. DOI: 10.1007/978-3-540-71431-6_17.

Black-Schaffer, D. (2003). Timing. EE183 Lecture 5. [Online]. Stanford, C.A , pp. 4-5. Available from: http://web.stanford.edu/class/ee183/handouts_spr2003/lecture5_spring2003.pdf.

Bolzoni, D. and Etalle, S. (2008). Approaches in Anomaly-based Network Intrusion Detection Systems. In: Di Pietro, R. and Mancini, V. L. eds. Intrusion Detection Systems. Advances in Information Security. Vol. 38. 1 edn. New York, NY, USA: Springer US, pp. 1-2. DOI: 10.1007/978-0-387-77265-3.

Boyer, R. S. and Moore, J. S. (1977). A Fast String Searching Algorithm. Communications of the ACM, 20(10), 762-772. DOI: 10.1145/359842.359859.

Bro (2013). Bro. [Online]. Last updated: 24 July 2013. Available from: http://www.bro.org/download/index.html [Accessed 24 July 2013].

Brodie, B. C., Taylor, D. E. and Cytron, R. K. (2006). A Scalable Architecture for High-Throughput Regular Expression Pattern Matching. ACM SIGARCH Computer Architecture News, 34(2), 191-202. DOI: 10.1145/1150019.1136500.

Chaves, R., et al. (2008). High Performance Embedded Architectures and Compilers; BRAM-LUT Tradeoff on a Polymorphic DES Design. , 4917, pp. 55-65. DOI: 10.1007/978-3-540-77560-7_5.

Chu, P. P. (2006). RTL Hardware Design using VHDL: Coding for Efficiency, Portability and Scalability. Vol. 1. New Jersey, USA: Wiley, pp. 1-15. ISBN-13: 978-0-471-72092-8.

Cisco (2013). IOS Intrusion Prevention System Deployment Guide. [Cisco IOS Intrusion Prevention System (IPS)] - Cisco Systems [Online]. Last updated: 24 July 2013. Available from: http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6586/ps6634/prod_white_paper0900aecd8062acfb.html [Accessed 24 July 2013].

Clark, C. R. and Schimmel, D. E. (2004). Scalable Pattern Matching for High Speed Networks. Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. Napa, CA, USA: IEEE, pp. 249-257. DOI: 10.1109/FCCM.2004.50.

Clark, C. R. and Schimmel, D. E. (2003). Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In: Cheung, P. Y. K. and Constantinides, G. A. eds. Field Programmable Logic and Application. Proceedings of the 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003 Lecture Notes in Computer Science. Vol. 2778. Lisbon, Portugal: Springer Berlin Heidelberg, pp. 956-959. DOI: 10.1007/978-3-540-45234-8_94.

Ficara, D., Giordano, S. and Procissi, G. (2008). An Improved DFA for Fast Regular Expression Matching. ACM SIGCOMM Computer Communication Review, 38(5), 29-40. DOI: 10.1145/1452335.1452339.

Floyd, R. W. and Ullman, J. D. (1982). The Compilation of Regular Expressions into Integrated Circuits. Journal of the ACM, 29(3), 603-622. DOI: 10.1145/322326.322327.

Ganegedara, T., Yang, Y. E. and Prasanna, V. K. (2010). Automation Framework for Large-Scale Regular Expression Matching on FPGA. Proceedings of the 2010 International Conference on Field Programmable Logic and Applications. Milano, Italy: IEEE, pp. 50-55. DOI: 10.1109/FPL.2010.21.

Gollmann, D. (2011). Computer Security. 3rd edn. Sussex, UK: Wiley. ISBN: 978-0-470-74115-3.

Gupta, P. and McKeown, N. (1999). Packet Classification on Multiple Fields. Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication - SIGCOMM '99. Massachusetts, USA: ACM SIGCOMM, pp. 147-160. DOI: 10.1145/316188.316217.

Hieu, T. T., et al. (2011). Optimization of Regular Expression Processing Circuits for NIDS on FPGA. Proceedings of the 2011 Second International Conference on Networking and Computing. Osaka, Japan: IEEE, pp. 105-112. DOI: 10.1109/ICNC.2011.23.

Hopcroft, J. E., Motwani, R. and Ullman, J. D., (2001). Introduction to Automata Theory, Languages and Computation. 2 edn. Boston, USA: Pearson/Addison-Wesley. ISBN: 0-201-44124-1.

Hutchings, B. L., Franklin, R. and Carver, D. (2002). Assisting Network Intrusion Detection with Reconfigurable Hardware. Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. Napa, CA, USA: IEEE, pp. 111-120. DOI: 10.1109/FPGA.2002.1106666.

IBM SPSS (2012). Released 2012. IBM SPSS Statistics for Windows, Version 21.0, 32-bit edn. Armonk, NY: IBM Corp.

IEEE (2001). IEEE Standard Verilog Hardware Description Language. [Online]. Last updated: 07 January 2004 [Accessed 01/10/2014].

Ilie, L., Solis-Oba, R. and Yu, S. (2005). Reducing the Size of NFAs by Using Equivalences and Preorders. In: Apostolico, A., Crochemore, M. and Park, K. eds. Combinatorial Pattern Matching. Proceedings of the 16th Annual Symposium, CPM 2005, Jeju Island, Korea, June 19-22, 2005. Lecture Notes in Computer Science. Vol. 3537. Jeju Island, Korea: Springer Berlin Heidelberg, pp. 310-321. DOI: 10.1007/11496656_27.

Ilie, L. and Yu Ilie, L. and Yu, S. (2002). Algorithms for Computing Small NFAs. In: Diks, K. and Rytter, W. eds. Mathematical Foundation of Computer Science 2002. Proceedings of the 27th International Symposium, MFCS 2002 Warsaw, Poland, August 26-30, 2002. Vol. 2420. Warsaw, Poland: Springer-Verlag Berlin Heidelberg, pp. 328-340. DOI: 10.1007/3-540-45687-2_27.

Jang, S., et al. (2009). SmartOpt: An Industrial Strength Framework for Logic Synthesis. Proceeding of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA '09. Monterey, CA, USA: ACM/SIGDA, pp. 237-240. DOI: 10.1145/1508128.1508165.

Jiang, W. and Prasanna, V. K. (2009). Field-Split Parallel Architecture for High Performance Multi-Match Packet Classification using FPGAs. Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures - SPAA '09. Calgary, Canada: ACM, pp. 188-196. DOI: 10.1145/1583991.1584044.

Jiang, W., Yang, Y. E. and Prasanna, V. K. (2010). Scalable Multi-Pipeline Architecture for High Performance Multi-Pattern String Matching. Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). Atlanta, GA, USA: IEEE, pp. 1-12. DOI: 10.1109/IPDPS.2010.5470374.

Kao, C. (2006). Benefits of Partial Reconfiguration Abstract. TechOnline [Online]. Last updated: 30 January 2006. Available from: http://www.techonline.com/electrical-engineers/education-training/tech-papers/4137820/Benefits-of-Partial-Reconfiguration [Accessed 21 June 2013].

Knuth, D. E., Morris, J. H. and Pratt, V. R. (1977). Fast Pattern Matching in Strings. SIAM Journal on Computing, 6(2), 323-350. DOI: 10.1137/0206024.

Komendantsky, V. (2012). Matching Problem for Regular Expressions with Variables. In: Loidle, H. and Pena, R. eds. Trends in Functional Programming. 13th International Symposium, TFP 2012, St, Andrews, UK, June, 12-14, 2012, Revised Selected Papers. Lecture Notes in Computer Science. Vol. 7829. St, Andrews, UK: Springer Berlin Heidelberg, pp. 149-166. DOI: 10.1007/978-3-642-40447-4_10.

Kruskal, J. B. (1956). On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. Proceedings of the American Mathematical Society, 7(1), 48-50. DOI: 10.1090/S0002-9939-1956-0078686-7.

Kumar, S., et al. (2006). Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. ACM SIGCOMM Computer Communication Review, 36(4), 339-350. DOI: 10.1145/1151659.1159952.

Kumar, S., Turner, J. and Williams, J. (2006). Advanced Algorithms for Fast and Scalable Deep Packet Inspection. Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems - ANCS '06. San Jose, CA, USA: ACM/IEEE, pp. 81-92. DOI: 10.1145/1185347.1185359.

Lacey, G., Taylor, G. W. and Areibi, S. (2016). Deep Learning on FPGAs: Past, Present, and Future. Cornell University Library [Online]. Available from: http://arxiv.org/abs/1602.04283 [Accessed 26/2/2016].

Lee, J., et al. (2007). A High Performance NIDS using FPGA-Based Regular Expression Matching. Proceedings of the 2007 ACM Symposium on Applied Computing - SAC '07. Seoul, Korea: ACM, pp. 1187-1191. DOI: 10.1145/1244002.1244259.

Lie, W. and Feng-yan, W. (2009). Dynamic Partial Reconfiguration in FPGAs. Proceedings of the 2009 Third International Symposium on Intelligent Information Technology Application. Nanchang, China: IEEE, pp. 445-448. DOI: 10.1109/IITA.2009.334.

Lin, C., et al. (2006). Optimization of Regular Expression Pattern Matching Circuits on FPGA. Proceedings of the Design Automation & Test in Europe Conference. Munich, Germany: IEEE, pp. 1-6. DOI: 10.1109/DATE.2006.244157.

Lin, C., Tai, Y. and Chang, S. (2007). Optimization of Pattern Matching Algorithm for Memory Based Architecture. Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems - ANCS '07. Orlando, Florida: ACM/IEEE, pp. 11-16. DOI: 10.1145/1323548.1323551.

Long, L. H., et al. (2011). ECEB: Enhanced Constraint Repetition Block for Regular Expression Matching on FPGA. ECTI Transactions on Electrical Engineering, Electronics, and Communications (ECTI-EEC), 9(1), 65-74.

Lysaght, P., et al. (2006). Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. Proceedings of the 2006 International Conference on Field Programmable Logic and Applications. Madrid, Spain: IEEE, pp. 1-6. DOI: 10.1109/FPL.2006.311188.

McNaughton, R. and Yamada, H. (1960). Regular Expressions and State Graphs for Automata. IEEE Transactions on Electronic Computers, 9(1), 39-47. DOI: 10.1109/TEC.1960.5221603.

Mealy, B. (2007). VHDL Tutorial. The Shock and Awe. Vol. 1. San Luis Obispo, CA, USA: Cal Poly State University, pp. 9-23.

Mitra, A., Najjar, W. and Bhuyan, L. (2007). Compiling PCRE to FPGA for Accelerating SNORT IDS. Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems - ANCS '07. Orlando, Florida: ACM/IEEE, pp. 127-136. DOI: 10.1145/1323548.1323571.

Mount, M. D. (2003). Lecture notes: Design and Analysis of Algorithms CMSC 451. 1 edn. Maryland, USA: Department of Computer Science, University of Maryland, pp. 57-59.

Moussalli, R., et al. (2014). High Performance FPGA and GPU Complex Pattern Matching over Spatio-temporal Streams. Geoinformatica, 19(2), 405-434. DOI: 10.1007/s10707-014-0217-3.

National Instruments (2011). Introduction to FPGA Hardware Concepts (FPGA Module) - LabVIEW 2011 FPGA Module Help. National Instruments [Online]. Last updated: 28 June 2013. Available from: http://zone.ni.com/reference/en-XX/help/371599G-01/lvfpgaconcepts/fpga_basic_chip_terms/ [Accessed 28 June 2013].

Perry, L. D. (2002). VHDL: Programming by Example. 4 edn. New York City: McGraw-Hill. DOI: 10.1036/0071409548.

Rayward-Smith, V. J. (1991). Automata Theory. In: McDermid, J. A. ed. Software Engineer''s Reference Book. [Online]. Vol. 1. Oxford: Butterworth Heineman, pp. 9/3-9/15. ISBN: 0750608137.

Rehman, U. R. (2003). Working with Snort Rules. In: Intrusion Detection with Snort: Advanced IDS Techniques using Snort, Apache, MySQL, PHP, and ACID. Bruce Perens' Open Source Series. Vol. 1. 4 edn. New Jersey, USA: Prentice Hall PTR, pp. 75-129. ISBN: 0-13-140733-3.

Remigiusz, W. (2008). Synthesis of Compositional Microprogram Control Units for Programmable Devices. Zielona, Góra: University of Zielona. PhD.

Roberts, L. G. and Wessler, B. D. (1970). Computer Network Development to Achieve Resource Sharing. Proceedings of the May 5-7, 1970, Spring Joint Computer Conference on - AFIPS '70 (Spring). ACM, Atlantic City, New Jersey, USA, pp. 543-549. DOI: 10.1145/1476936.1477020.

Roesch, M. (1999). Snort - Lightweight Intrusion Detection for Networks. Proceedings of the 13th USENIX Conference on System Administration, LISA '99. Seattle, Washington, USA: USENIX Association Berkeley, pp. 229-238.

Rohatgi, V. K. and Saleh, E. M. A. K. (2001). An Introduction to Probabilty and Statistics. Wiley Series in Probability and Statistics. Vol. 1. 2 edn. Delhi, India: John Wiley and Sons. ISBN: 978-81-265-1926-2.

Rufi, A. W. (2006). Network Security 1 and 2 Companion Guide (Cisco Networking Academy Program) (Companion Guide) . 1 edn. Indianapolis, Indiana, USA: Pearson Education, Cisco Press, pp. 17-39. ISBN-10: 1-58713-162-5.

Sidhu, R. and Prasanna, V. K. (2001). Fast Regular Expression Matching using FPGAs. Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines - FCCM '01. Rohnert Park, CA, USA: IEEE, pp. 227-238. DOI: 10.1109/FCCM.2001.22.

Singapura, S. G., et al. (2015). 1. FPGA Based Accelerator for Pattern Matching in YARA Framework. Technical Report. Computer Engineering Technical Report. Los Angeles, CA, USA: Computer Engineering, Ming Hsieh Department of Electrical Engineering-Systems. Report number: CENG-2015-05. pp. 1-10.

Singh, D., et al. (2016). Collaborative IDS Framework for Cloud. International Journal of Network Security, 18(4), 699-709.

Sourcefire (2013). SNORT Users Manual 2.9.7 [Online]. Available from: http://manual.snort.org/snort_manual.html [Accessed 1/23/2016].

Snort (2013). Snort IDS/Rules. SNORT [Online]. Last updated: 18 July 2013. Available from: http://www.snort.org/ [Accessed 18 July 2013].

Sourcefire (2009). SNORT User's manual. The Snort Project. 2.8.5 edn. SNORT, pp. 7-9.

Sourdis, I. and Pnevmatikatos, D. (2004). Pre-Decoded CAMs for Efficient and High-Speed NIDS Pattern Matching. Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. Napa, California: IEEE, pp. 258-267. DOI: 0-7695-2230-0.

Sourdis, I. and Pnevmatikatos, D. (2003). Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System. Field Programmable Logic and Application, 2778, 880-889. DOI: 10.1007/978-3-540-45234-8_85.

Sutton, P. (2004). Partial Character Decoding for Improved Regular Expression Matching in FPGAs. Proceedings of the 2004 IEEE International Conference on Field- Programmable Technology (IEEE Cat. no.04EX921). Brisbane, Australia: IEEE, pp. 25-32. DOI: 10.1109/FPT.2004.1393247.

Symantec (2007). W32.SQLExp.Worm.W32.SQLExp.Worm|Symantec [Online]. Last updated: 13 February 2007. Available from: http://www.symantec.com/security_response/writeup.jsp?docid=2003-012502-3306-99; [Accessed 10/17/2014].

Tan, L. and Sherwood, T. (2005). A High Throughput String Matching Architecture for Intrusion Detection and Prevention. Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05). Madison, Wisconsin, USA: IEEE, pp. 112-122. DOI: 10.1109/ISCA.2005.5.

Thompson, K. (1968). Programming Techniques: Regular Expression Search Algorithm. Communications of the ACM, 11(6), 419-422. DOI: 10.1145/363347.363387.

Trefzer, M. A., et al. (2015). Fighting Stochastic Variability in a D-type Flip-Flop with Transistor-Level Reconfiguration. IET Computers & Digital Techniques, 9(4), 190-196. DOI: 10.1049/iet-cdt.2014.0146.

Tripp, G. (2005). A Finite-State-Machine Based String Matching System for Intrusion Detection on High-Speed Networks. In: Turner, P. and Broucek, V. eds. 14th EICAR 2005 Conference Best Paper Proceedings. EICAR, pp. 26-40. ISBN: 87-987271-7-6.

Tripp, G. (2006). A Parallel String Matching Engine for use in High Speed Network Intrusion Detection Systems. Journal in Computer Virology, 2(1), 21-34. DOI: 10.1007/s11416-006-0010-4.

Tripp, G. (2008). Regular Expression Matching with Input Compression and Next State Prediction. Technical Report. Kent Academic Repository. Canterbury: University of Kent Press. Report number: 3-08.

Wain, R., et al. (2006). An overview of FPGAs and FPGA programming; Initial experiences at Daresbury. Computational Science and Engineering Department. Vol. 1. 2 edn. Daresbury, Cheshire, UK: CCLRC, pp. 4-5. ISSN: 1362-0207.

Wang, H., et al. (2010). A Modular NFA Architecture for Regular Expression Matching. Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA'10. Monterey, CA, USA: ACM/SIGDA, pp. 209-218. DOI: 10.1145/1723112.1723149.

Wellman, B. (2001). Computer Networks as Social Networks. Science, 293(5537), 2031-2034.DOI: 10.1126/science.1065547.

WinterGreen Research Inc. (2010). 445. Programmable Logic IC Market Shares and Forecasts, Worldwide, 2010 - 2016. Technical Report. Report No. SH245544713. Massachusetts, USA: WinterGreen.

Xilinx (2012c). Virtex-6 FPGA Libraries Guide for Schematic Designs UG624. Vol. 14.3. Xilinx, Inc, pp. 9-268.

Xilinx (2012b). XST User G FPGA Guide for Vixtex-6 FPGA, Spartan-6, and 7 Series Devices UG687. Vol. 14.1. Xilinx, Inc, pp. 13-466.

Xilinx (2012a). Virtex-6 FPGA Configurable Logic Block UG695. Vol. 1.2. Xilinx, Inc, pp. 7-11.

Xilinx (2011). Block RAM Resources UG363. Vol. 1.6. Xilinx, Inc, pp. 11-13.

Xilinx (2010). ISE In-Depth Tutorials UG695. Vol. 12.3. Xilinx, Inc, pp. 96-150.

Xilinx (2009). Configuring or Programming a Target Device. [Online]. Last updated: 2009. Available from: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/pp_p_process_configure_target_device.htm [Accessed 25 September 2013].

Xilinx (2008b). Implementation Overview for FPGAs. [Online]. Last updated: 2008b. Available from: http://www.xilinx.com/itp/xilinx10/isehelp/ise_c_implement_fpga_design.htm [Accessed 25 September 2013].

Xilinx (2008a). XST User Guide. Vol. 10.1. Xilinx, Inc, pp. 314-503.

Xilinx (1999). VHDL Reference Guide. Xilinx Development System. Xilinx, Inc, pp. 2(1-28).

Xilinx ISE (2012). Xilinx ISE Project Navigator Application Version P.49d [Software]. Vol. 14.4 (nt64). Xilnx Inc.

Yamagaki, N., Sidhu, R. and Kamiya, S. (2008). High-Speed Regular Expression Matching Engine using Multi-Character NFA. Proceedings of the 2008 International Conference on Field Programmable Logic and Applications. Heidelberg, Germany: IEEE, pp. 131-136. DOI: 10.1109/FPL.2008.4629920.

Yang, Y. E., Jiang, W. and Prasanna, V. K. (2008). Compact Architecture for High-Throughput Regular Expression Matching on FPGA. Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems - ANCS '08. San Jose, CA, USA: ACM/IEEE, pp. 30-39. DOI: 10.1145/1477942.1477948.

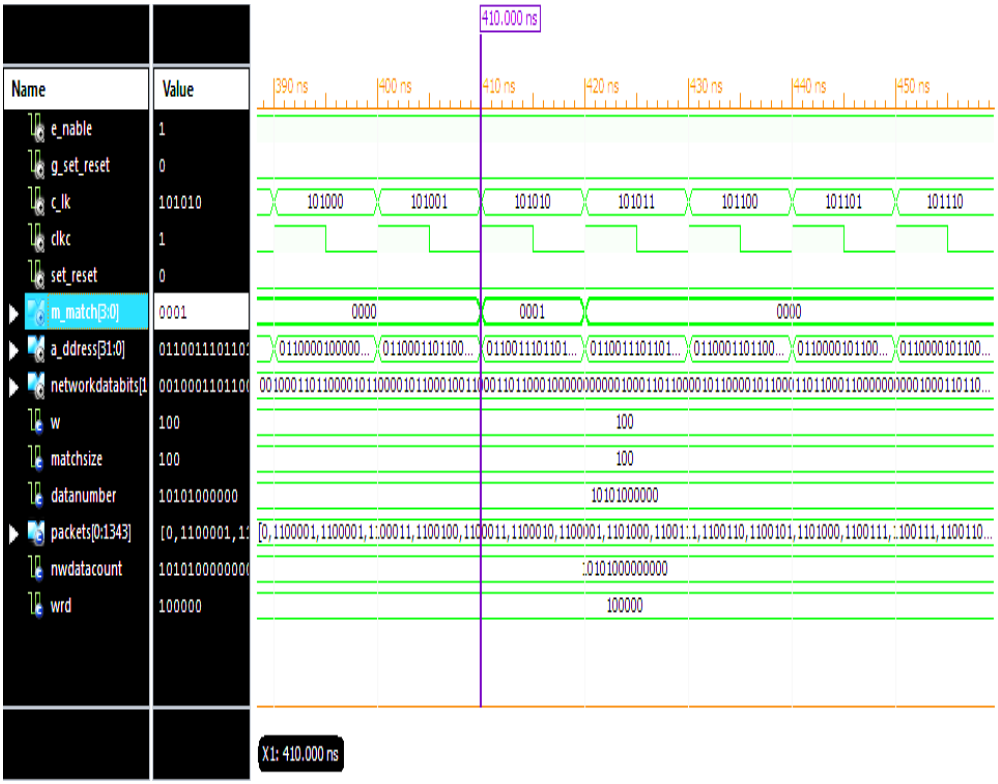
Yang, Y. E. and Prasanna, V. K. (2009). Software Toolchain for Large-Scale RE-NFA Construction on FPGA. International Journal of Reconfigurable Computing, 2009(2), 1-10. DOI: 10.1155/2009/301512.

Yang, Y. and Prasanna, V. (2012). High-Performance and Compact Architecture for Regular Expression Matching on FPGA. IEEE Transactions on Computers, 61(7), 1013-1025. DOI: 10.1109/TC.2011.129.

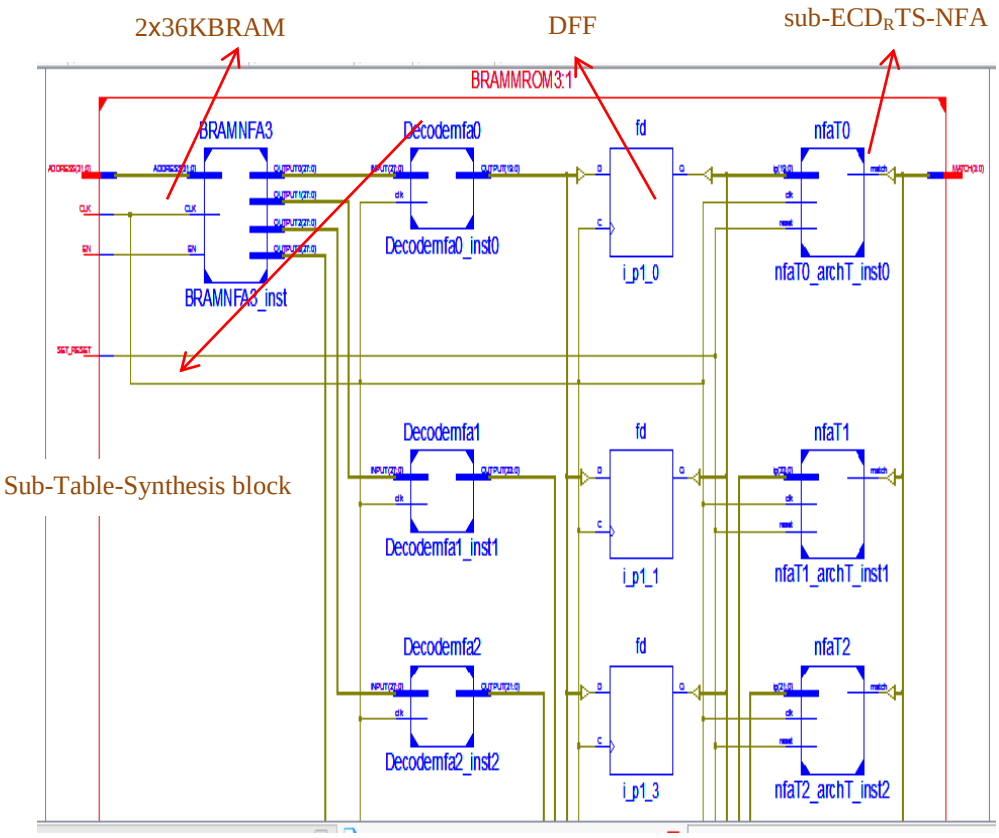
Yu, F., et al. (2006). Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection. Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems - ANCS '06. San Jose, CA, USA: ACM/IEEE, pp. 93-102. DOI: 10.1145/1185347.1185360.

Zhang, Y. and Lee, W. (2000). Intrusion Detection in Wireless Ad-Hoc Networks. Proceedings of the 6th Annual International Conference on Mobile Computing and Networking. Boston, MA, USA: ACM, pp. 275-283. DOI: 10.1145/345910.345958.

APPENDIX 1.1: Simulated 4-byte Matching BG2RE sub-REME.



APPENDIX 1.2: RTL Diagram for BG2RE sub-REME.



APPENDIX 1.3: Results for the Ten BG5RE sub-REMEs.

BG5RE	Number of States	Number of LUTs	Speed	LUTs /State	Average Throughput
Engine	NOS	NOL	MHz	NoL/ NoS	AverageTp
1	1323	2960	252.91	2.24	7.9
2	1151	1352	282.65	1.175	8.83
3	1736	1999	255.76	1.152	7.99
4	485	2913	263.64	6.006	8.24
5	925	3577	255.17	3.867	7.98
6	1492	2020	294.64	1.354	9.21
7	1465	2339	262.19	1.597	8.19
8	882	1966	271.67	2.229	8.49
9	648	2489	250.88	3.841	7.84
10	1020	4327	253.68	4.24	7.93
Std.Dev	395.7	875.71	14.52	1.64	0.46
Mean	1112.7 [a]	2594.2 [b]	264.32	2.77	8.26
AvgTp	8.26 [c]				
AvgTpe = (a/b)*c)	3.54				

APPENDIX 1.4: Results for the Ten BG4RE sub-REMEs.

BG4RE	Number of States	Number of LUTs	Speed	LUTs /State	Average Throughput
Engine	NOS	NOL	MHz	NoL/ NoS	AverageTp
1	1161	1299	321.75	1.12	10.05
2	1811	1834	284.41	1.01	8.89
3	802	633	392.47	0.79	12.27
4	703	1818	263.85	2.59	8.25
5	536	1628	285.63	3.037	8.93
6	765	1299	286.7	1.698	8.96
7	870	1377	268.6	1.572	8.39
8	1487	1454	281.29	0.98	8.79
9	1114	1905	300.21	1.71	9.38
10	1038	2249	252.14	2.171	7.88
Std.Dev	386.13	444.63	39.71	0.75	1.25
Mean	1028.7 [a]	1549.6 [b]	293.71	1.67	9.18
AvgTp	9.18 [c]				
AvgTpe = (a/b)*c)	6.09				

APPENDIX 1.5: Results for the Ten BG3RE sub-REMEs.

BG3RE	Number of States	Number of LUTs	Speed	LUTs /State	Average Throughput
Engine	NOS	NOL	MHz	NoL/ NoS	AverageTp
1	752	820	299.85	1.09	9.37
2	605	630	362.19	1.04	11.31
3	886	1080	303.31	1.219	9.48
4	230	1197	307.6	5.204	9.61
5	411	1462	281.37	3.557	8.79
6	2761	1082	312.79	0.392	9.78
7	1320	879	321.23	0.666	10.04
8	1266	1707	309.7	1.348	9.68
9	1198	1748	271.44	1.459	8.48
10	800	1139	310.95	1.424	9.72
Std.Dev	708.57	368.76	24.23	1.49	0.76
Mean	1022.9 [a]	1174.4 [b]	308.04	1.74	9.63
AvgTp	9.63 [c]				
AvgTpe = (a/b)*c)	8.39				

APPENDIX 1.6: Results for the Ten BG2RE engines.

BG2RE	Number of States	Number of LUTs	Speed	LUTs/ State	Average Throughput
Engine	**NOS**	**NOL**	**MHz**	**NoL/ NoS**	**AverageTp**
1	925	242	407.5	0.262	12.73
2	1300	618	386.55	0.475	12.07
3	319	637	362.19	1.997	11.32
4	875	356	380.19	0.407	11.88
5	526	715	291.38	1.359	9.1
6	782	266	375.78	0.34	11.74
7	692	457	370.92	0.66	11.59
8	1266	345	406.5	0.273	12.7
9	337	584	363.5	1.733	11.36
10	1252	463	328.84	0.37	10.28
Std.Dev	367.88	165.07	35.06	0.66	1.1
Mean	**827.4**[a]	**468.3**[b]	**367.34**	**0.79**	**11.48**
AvgTp	**11.48**[c]				
AvgTpe = (a/b)*c)	**20.28**				

APPENDIX 1.7: Summary of the Average Throughput per REME Design.

Engines	BG5RE	BG4RE	BG3RE	BG2RE
1	7.9	10.05	9.37	12.73
2	8.83	8.89	11.31	12.07
3	7.99	12.27	9.48	11.32
4	8.24	8.25	9.61	11.88
5	7.98	8.93	8.79	9.1
6	9.21	8.96	9.78	11.74
7	8.19	8.39	10.04	11.59
8	8.49	8.79	9.68	12.7
9	7.84	9.38	8.48	11.36
10	7.93	7.88	9.72	10.28
Std.Dev	0.454	1.242	0.756	1.095
Mean	8.26	9.179	9.626	11.477

APPENDIX 1.8: Summary of the ratio of LUTs per State in each REME Design.

Engines	BG5RE	BG4RE	BG3RE	BG2RE
1	2.24	1.12	1.09	0.262
2	1.175	1.01	1.04	0.475
3	1.152	0.79	1.219	1.997
4	6.006	2.59	5.204	0.407
5	3.867	3.037	3.557	1.359
6	1.354	1.698	0.392	0.34
7	1.597	1.572	0.666	0.66
8	2.229	0.98	1.348	0.273
9	3.841	1.71	1.459	1.733
10	4.24	2.171	1.424	0.37
Std.Dev	1.637	0.743	1.483	0.655
Mean	2.7701	1.6678	1.7399	0.7876